CORRECTED VERSION*

PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 :		(11) International Publication Number:	WO 96/15501
G06F 17/30	A1	(43) International Publication Date:	23 May 1996 (23.05.96)

(21) International Application Number: PCT/US95/15028

(22) International Filing Date: 13 November 1995 (13.11.95)

(30) Priority Data:

08/339,481	10 November 1994 (10.11.94)	US
08/527,161	12 September 1995 (12.09.95)	US

(71) Applicant: CADIS, INC. [US/US]; 1909 26th Street, Boulder, CO 80302 (US).

(71)(72) Applicants and Inventors: KAVANAGH, Thomas, S. [US/US]; 65 Bellevue Drive, Boulder, CO 80302 (US). BEALL, Christopher, W. [US/US]; 679 Cougar Drive, Boulder, CO 80302 (US). HEINZ, William, C. [US/US]; 8256 Johnson Court, Arvada, CO 80005 (US). MOTYCKA, John, D. [US/US]; 30130 Chestnut Drive, Evergreen, CO 80439 (US). PENDLETON, Samuel, S. [US/US]; 976 West Dahlia Street, Louisville, CO 80027 (US). SMALLWOOD, Thomas, D. [US/US]; 308 Morning Star Lane, Lafayette, CO 80026 (US). TERPENING, Brooke, E. [US/US]; 25221 Westridge Road, Golden, CO 80403 (US). TRAUT, Kenneth, A. [US/US]; 4151 Cooper Court, Boulder, CO 80303 (US).

(74) Agent: LEACH, Sydney; Baker & McKenzie, Suite 4500, 2001 Ross Avenue, Dallas, TX 75201 (US).

(81) Designated States: AT, AU, BB, BG, BR, BY, CA, CH, CN, CZ, DE, DK, ES, FI, GB, HU, JP, KP, KR, KZ, LK, LU, LV, MG, MN, MW, NO, NZ, PL, PT, RO, RU, SD, SE, SK, UA, UZ, VN, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published

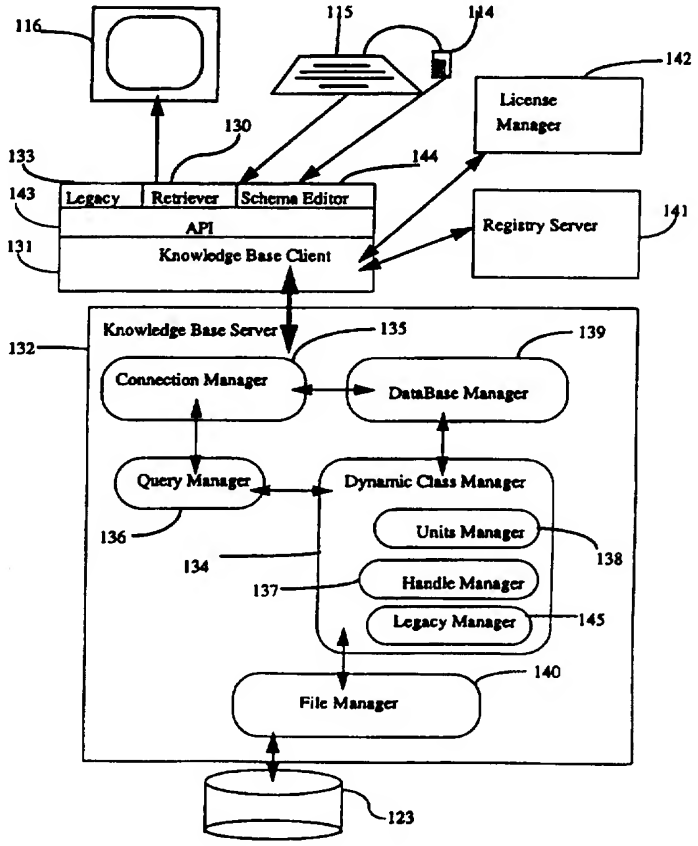
With international search report.

Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: OBJECT ORIENTED DATABASE MANAGEMENT SYSTEM

(57) Abstract

The present invention provides a method and apparatus for an object oriented database management system. The present invention may be advantageously used in a client/server architecture comprising a knowledge base client and a knowledge base server (132). A plurality of users may access the system at the same time. In a preferred embodiment, the knowledge base server (132) may include a dynamic class manager (134), a connection manager (135), a query manager (136), a handle manager (137), a units manager (138), a database manager (1349), and a file manager (140). The object oriented database system is hierarchical. Each instance in a knowledge base may be a member of a class, and a class may be a subclass of a parent class, and so on.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria
AU	Australia
BB	Barbados
BE	Belgium
BF	Burkina Faso
BG	Bulgaria
BJ	Benin
BR	Brazil
BY	Belarus
CA	Canada
CF	Central African Republic
CG	Congo
CH	Switzerland
CI	Côte d'Ivoire
CM	Cameroon
CN	China
CS	Czechoslovakia
CZ	Czech Republic
DE	Germany
DK	Denmark
ES	Spain
FI	Finland
FR	France
GA	Gabon
GB	United Kingdom
GE	Georgia
GN	Guinea
GR	Greece
HU	Hungary
IE	Ireland
IT	Italy
JP	Japan
KE	Kenya
KG	Kyrgystan
KP	Democratic People's Republic of Korea
KR	Republic of Korea
KZ	Kazakhstan
LI	Liechtenstein
LK	Sri Lanka
LU	Luxembourg
LV	Latvia
MC	Monaco
MD	Republic of Moldova
MG	Madagascar
ML	Mali
MN	Mongolia
MR	Mauritania
MW	Malawi
NE	Niger
NL	Netherlands
NO	Norway
NZ	New Zealand
PL	Poland
PT	Portugal
RO	Romania
RU	Russian Federation
SD	Sudan
SE	Sweden
SI	Slovenia
SK	Slovakia
SN	Senegal
TD	Chad
TG	Togo
TJ	Tajikistan
TT	Trinidad and Tobago
UA	Ukraine
US	United States of America
UZ	Uzbekistan
VN	Viet Nam

OBJECT ORIENTED DATABASE MANAGEMENT SYSTEM

BACKGROUND OF THE INVENTION

The present invention relates to an object oriented database management system that is optimized for fast reads of the data, and is particularly suited for applications where extensive updating is not necessary. The database is structured so that when an item does not have a value, nothing is stored. Therefore, memory space is not wasted storing null values, and search speed is improved because no time is consumed searching such null fields.

While various features of the present invention may be advantageously used in other applications, the invention is described herein with reference to the problem of managing parts and components in a manufacturing operation. The invention is particularly useful for solving problems in parts management which have existed for a long time.

Often, the competitive success of a manufacturing company may largely depend upon the company's ability to bring products to market faster. The rewards for an enterprise that is able to achieve this objective may be considerable. The penalty for failing to achieve this objective can be the loss of a customer or even an entire market. In a typical company, re-engineering or redesigning the parts selection process may significantly improve the operations of the company and achieve major gains in bringing products to market faster. In addition, significant cost savings may be achieved.

Design engineering has been the focal point in the competitive drive to get products to market quicker, at reduced cost, and with improved quality. Companies are continually striving to make the design engineer more efficient. This quest for efficiency translates into providing more effective tools for the design activity thereby making the design activity a larger portion of the design day.

The culmination of every design cycle in a manufacturing company is the parts selection process that will result in a completed bill of materials. Design engineers typically have to specify and select dozens of components that will satisfy their design requirements. In every case, the design engineer will be

presented with a choice that collectively can have a major strategic impact on the firm. The implicit choice that each design engineer faces when specifying and selecting every part is the question of whether he or she can re-use an existing part, or whether he or she needs to release a new part.

Depending on the design engineer's answer to this deceptively simple question, the company may be required to support a new part at considerable cost. Market research has shown that design engineers in large companies find it easier to add a new part, even if an exact match or acceptable substitute already exists, because it takes too much time and trouble to look up parts to determine whether an existing part would be suitable. However, releasing even one new part is time consuming and expensive. **Figure 1** summarizes the typical process that a manufacturing company goes through prior to releasing a new part.

If an existing approved part can be used in a design, the design engineer has more time for design, the expensive process of releasing a new part is avoided, and the value of the prior part release process will be maximized. However, what is required is a quick and easy way to locate parts for use. When a part is quickly found and used, not only does the designer benefit, but the design engineering department, procurement, manufacturing, field service, and every other group downstream from the design engineer will benefit as well. Typically though, after spending too much valuable design time looking for a released part and not finding it, the design engineer simply specifies another new part.

The reason the designer often has difficulty finding parts is because most systems which reference parts do so by part number. The designer knows the functional attributes, geometric description, and other characteristics of what is required, but rarely knows the part number from which the correct part can be identified. Efforts to address this problem in the past have made part descriptions available through key-words; but oftentimes the descriptions are not standardized and are usually limited to a bare bones description due to restricted field lengths. In the past, accurate access by design engineers to released parts information has usually been inadequate.

Because of the need to use existing parts, ad-hoc crutches

have been developed by many organizations. These include intelligent part numbering systems, crib-sheets, "where used" systems derived from bill of materials, group technology, CAD drawing management systems, and occasionally description driven RDBMS applications. These solutions are ad-hoc because:

1. These crutches are not complete solutions; they often lead to the circumvention of the existing part selection and release process in order to get the job done.
2. They are based on tools that are designed for other primary tasks and are typically inefficient or are misused in this application.
3. The organization develops and applies resources not directly related to the focus of the business.
4. Too many people become part selection experts on their current design focus only, limiting mobility of personnel to new projects.
5. There are no tools that measure the frequency of finding a suitable part, or provide any measure of redundancy between the newly released parts with those already available for use.
6. Inevitably these attempts to develop a complete system are unsatisfactory and are abandoned.

There is an additional reason why these past attempts to address this problem cannot be characterized as complete solutions. They do not adequately address the company's entire pool of released parts. This parts pool typically has characteristics that hinder comprehensive management and which have stifled full corrective action by any existing system. Such characteristics include the fact that the data tends to be widely scattered across the company, and among many different systems. Most parts are poorly described, and some parts can never be found because of description inconsistencies. There are many similar parts -- parts that are different, but which would satisfy the same design criteria if the parts could be identified. The pool of parts almost never shrinks. Typically, no matter how big it is today, it will be bigger tomorrow.

In the past, ad-hoc solutions invariably attempted to

address the problem by utilizing key-word search tools. Searches on user specified key-words are part of many relational database applications. A key-word query on a relational database typically causes the database to search a specific table for some text-string. These applications may support wild cards, the option for case sensitivity, or other functions associated with the key-word match. However, given the inconsistencies in typical part descriptions noted above, key-word approaches have been severely limited in their effectiveness. In a database that supports key-word searches, a question is posed in terms of key-words and answers are returned, but it is never known if all possible answers are returned. In a parts management system, it is critical to find all items and all related or similar items in a database. Otherwise, there can be no assurance that a suitable part does not already exist in the company's database, and the cost of creating a new part to add to the existing database may be incurred unnecessarily when the system fails to find suitable existing parts.

The example shown in **Table 1** highlights the limitation of a key-word based parts retrieval system. The four entries shown in **Table 1** represent examples of typical entries in a parts database. An elementary key-word search on the term 'cam follower bearing' would likely return with only one part found, #0002. A sophisticated system might return with three partial matches, finding some of the terms in #0001, #0002, and #0003. It is unlikely that a system would know that 'track roller' is a synonym for 'cam follower.' Also, roller and needle are sometimes synonyms. A search involving 'inch' would only find a match in the first listed part. Finally, while the last two parts imply they are bearings, the descriptions do not explicitly identify them as such.

Table 1

Part #	Description
0001	Bearing, cam follower, roller, 1.0 inch
0002	Cam follower bearing, needle, 1.0 "
0003	Cam follower, 1.0", roller
0004	Track roller , 1.0"

Problems have arisen in the past because it is not uncommon for various units to be specified differently in typical part descriptions. For example, some parts may have temperature characteristics specified in degrees Centigrade, while others may be specified in Fahrenheit. Also, one screw may have a length that is expressed as 1 inch. Another screw may have a length expressed as 2.5 centimeters. Both screws may be acceptable substitutes for the same design requirement. However, prior database management tools have not been able to satisfactorily deal with units in a way which would allow both parts to be found in response to a search for existing parts having a range of lengths that included both 1 inch and 2.5 centimeters.

The standard relational database management systems (RDBMS) model is unsatisfactory for developing a parts management solution. Internally developed corporate systems have inevitably been built on a standard RDBMS technology and, in general, have not been satisfactory to the end-user.

In an effort to deal with these problems, some companies have developed a dollar cost estimate of the release process, and have provided a mechanism to charge it back to the design engineering group. The rationale behind such an approach is that the design engineer is the only person who can influence the outcome one way or another. By choosing to specify a new part, the design engineer commits the company to a series of process steps, such as those shown in **Figure 1**, that subsequently result in time and cost incurrence to the corporation. However, such efforts have been less than satisfactory, and serve to demonstrate the need for a quicker and easier system for looking up parts to determine whether an existing part would be suitable.

It would be desirable to entirely eliminate triggering the process depicted in **Figure 1** by avoiding the release of a new part whenever possible. In the past, design engineers have not been provided with the needed tools to specify and select parts that have already been released. There has been a need to provide an approach which would allow a company to avoid duplicating process costs (in time and effort) that have been incurred earlier in releasing a part, if an existing part is an exact match to the design requirements or an acceptable substitute.

A company's pool of existing parts data is potentially a valuable asset, but its effective value is discounted by the above-described characteristics which inhibit the data from being a useful and readily available resource of prior company knowledge and investment. Therefore, any solution that can affordably transform this pool of existing parts data into a useful information resource would be of great value to a company. However, effective tools to manage it have not been available in the past.

SUMMARY OF THE INVENTION

The present invention, in its preferred embodiment, may include a retriever means, a knowledge base client means, and a knowledge base server means. A legacy means is preferably included to facilitate organization of an existing legacy database into a schema for use in connection with the present invention. In a preferred embodiment, the knowledge base server means includes a dynamic class manager means, a connection manager means, a query manager means, a handle manager means, a units manager means, a database manager means, and a file manager means. A preferred system also includes a registry server means and license manager means to control unauthorized user access to the system.

The present invention may be used to provide a part management system which has a number of advantageous features. A system in accordance with the present invention provides a tool for design engineers which enables them to intuitively, definitively, and virtually instantaneously find a released part that is either an exact match or an acceptable substitute for the design requirements, if such a part exists. Duplicate parts can be eliminated, and inventory carrying costs reduced as well.

Through the use of an object oriented knowledge base, the present invention can make access to part data intuitive, instantaneous, definitive, and can encompass all parts. The present system can transform a company's poorly managed pool of existing parts data into a valued corporate asset. It can provide ongoing consistency and control to the specification, release, and subsequent retrieval of all parts information.

Part classes, sub-classes, part characteristics such as

shape, material, and dimensions, among others, fit very well within the object oriented environment of the present invention. Parts are treated as objects within a parts family or "**schema**".

The present invention uses attribute searches, which offer decided advantages over generic key-word searches. The incomplete search problems associated with key-word matching which are described above with reference to **Table 1** may be solved when the same data is restructured as parametric attributes. A parametric attribute description consists of (1) reducing all terminology to some standard form, (2) describing each term as some value of an attribute related to an object or subclass, and (3) ordering the set of attributes of the object. In this case, cam follower bearings are classified under subclass of bearings called "mounted bearings". This is illustrated in **Table 2**. Described this way, the parts are easily related, and appear to correspond to the same part. This would not be apparent from a key-word search.

Table 2

Part #	Ojbect	Mounting Type	Element Type	Diameter inches
0001	mounted bearing	cam follower	roller	1.0
0002	mounted bearing	cam follower	roller	1.0
0003	mounted bearing	cam follower	roller	1.0
0004	mounted bearing	cam follower	roller	1.0

In the present invention, users search a parts database by selecting attributes that describe a part. Selection consists of sifting from general to detailed part attributes. All possible questions are linked to the attributes; the user merely selects from the enumerated possibilities. This sifting mechanism has the effect of masking unwanted parts. The intent is to leave parts that exactly fit the search criteria, but not eliminate any parts that *might* fit.

The present invention is an effective, on-going part specification, description, and retrieval system. Parts are found by describing them using their relevant attributes.

Attributes can be both parametric (length, capacitance, etc.) and non-parametric (cost, preferred, etc.). The description process is intuitive to the occasional user and does not require specialized computer expertise. Needed parts may be found virtually instantly. This level of performance encourages widespread usage of the system. The response time is essentially independent of the size of the database searched and of the number of users at any point in time.

A system in accordance with the present invention provides definitive access to the data. If a needed part exists, the user will be able to find it. If a part does not exist, the user will know that too with certainty, so that a new part can be released with confidence. The system is capable of retrieving all parts fitting the description criteria completely, as well as all parts that closely match or satisfy a subset of the criteria. The system facilitates the selection of parts based on preferred attributes. Examples of preferred attributes include: "standard" values (which encourage design standardization), low field failure rates (which ensures reliability), low unit cost, and preferred suppliers.

The present system can affordably transform a company's pool of existing parts legacy data into usable information. The present system enables a design engineer to painlessly create and edit descriptions of parts based on critical engineering attributes. All part descriptions may be standardized in terms of content and format as a function of the type of part. The descriptions are independent of arbitrary and pre-determined field length limitations, and are able to automatically accommodate the varying field length requirements of different part types. The system is flexible in that it may be easily modified to accommodate major changes triggered by internal or external realities. This includes addition and deletion of entire part families, new product lines, corporate consolidations, mergers, and acquisitions.

The present system provides unit measure convertibility. The user is able to specify a part in his or her unit-of-measure of choice. The system provides rules governing the conversion of units-of-measure of parts. For some part families convertibility of units is allowed and required, for others,

convertibility is prohibited; the system knows what rules apply to which part families.

The present system provides an open system environment with connectivity to any other application or system across the enterprise. Enterprise-wide desktop access to all parts information is provided. Part information on newly specified parts is instantly available throughout the corporation. The elimination of the information time lag between engineering and other departments involved in parts management fosters concurrent engineering practices. The system also provides management and control functions associated with the release of parts into the system.

The present system enables design engineers, and other users, to locate parts by describing them in terms of parametric and non-parametric attributes. It supports dynamic management (additions, deletions, and manipulations) of part families and attributes to accommodate both standard and proprietary parts. It provides on-going structure, consistency, and control in the management of the part specification and description process. It also includes the company's existing (legacy) parts in the on-going system.

The present invention may be advantageously used in a client/server architecture comprising a knowledge base client and a knowledge base server. The present invention provides a particularly advantageous concurrency control mechanism for an object oriented database management system that is read oriented. In a preferred embodiment, the knowledge base server includes an object oriented lock manager, a dynamic class manager, a connection manager, a query manager, a handle manager, a units manager, a database manager, and a file manager.

The object oriented lock manager of the present invention may be used to provide a concurrency control mechanism which has a number of advantageous features. A system in accordance with the present invention maximizes availability of a tool for design engineers. The invention provides optimal availability by allowing users to query and view class objects without disruption of their view while modifications are being made by other users. These modifications would preferably include additions, deletions, and edits of classes, attributes, instances, and

parameters.

The invention optimizes performance of the concurrency control system by using lock inheritance based on class objects. The lock manager means implements a mechanism for locks to be placed on a class without subclass inheritance of the lock. This mechanism is a class lock. The lock manager means also provides an inheritance mechanism for locks. The inheritance mechanism is a tree lock. Tree locking a class will lock all descendants of that class by inheritance without physically requiring the placement of class locks on the descendant classes. The present invention employs true share locks and exclusive locks. The present invention also provides a novel implementation of a lock mode that is a hybrid between a share lock and an exclusive lock, which is referred to as an "update" lock.

The invention optimizes performance by simplifying the number of objects that need to be locked by using class level lock granularity. The granularity or scope of a class lock is the class itself, the attributes defined by the class, and the instances associated with that class. The present invention does not allow an instance to be locked independently of the class to which it belongs.

The knowledge base client means uses the object oriented lock means mechanisms to place locks of appropriate granularity and inheritance to provide the maximum availability, stability, and performance of a tool using these means.

Further features and advantages of the present invention will be appreciated in connection with the drawings and the following detailed description of a presently preferred embodiment.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a flow chart depicting a typical conventional parts management process.

Figure 2 is a diagram of a typical network environment that is suitable for use in connection with the present invention.

Figure 3 is a block diagram depicting an overall architecture for a system according to the present invention.

Figures 4A and 4B represent a flow chart showing a login procedure for accessing the system.

Figure 5 depicts an initial display screen showing the part specification window.

Figure 6 depicts an example of the part specification window during a search.

Figure 7 is a flow chart for selecting a class.

Figure 8 is a flow chart depicting the procedure for updating the part count and display.

Figure 9 is a flow chart depicting the procedure for opening a class.

Figure 10 depicts a display screen showing information displayed in the part specification window.

Figure 11 is a flow chart depicting the procedure for closing an open class.

Figure 12 is a flow chart depicting the procedure for selecting text search criteria.

Figure 13 depicts a display screen showing information displayed in the part specification window.

Figure 14 is a flow chart depicting the procedure for selecting numeric search criteria.

Figure 15 depicts a custom numeric dialog box.

Figure 16 depicts a display screen showing information displayed in the part specification window.

Figure 17 is a flow chart depicting the procedure for selecting boolean search criteria.

Figure 18 depicts a display screen showing information displayed in the part specification window.

Figure 19 is a flow chart depicting the procedure for selecting enumerated search criteria.

Figure 20 depicts a display screen showing information displayed in the part specification window.

Figure 21 depicts a display screen showing information displayed in the part specification window.

Figure 22 is a flow chart depicting the procedure for selecting attribute order for display.

Figure 23 is a flow chart depicting the procedure for displaying search results.

Figure 24 depicts a display screen showing information displayed in the search results window.

Figure 25 is a flow chart depicting the procedure for doing a

query.

Figure 26 is a flow chart depicting the procedure for displaying part information.

Figure 27 depicts a display screen showing information displayed in the part information window.

Figure 28 is a flow chart depicting the procedure for launching a user action.

Figure 29 depicts a display screen showing an example of a user action launched by the procedure depicted in **Figure 28**.

Figure 30 is a flow chart depicting the procedure followed when the user actuates the apply button.

Figure 31 depicts a display screen showing information displayed in the part specification window.

Figure 32 is a flow chart depicting the procedure followed when the user actuates the edit button.

Figure 33 is a flow chart depicting the procedure followed when the user actuates the sort button.

Figure 34 depicts a display screen showing information displayed in the sort dialog box.

Figure 35 is a flow chart depicting procedures followed when a user edits parts.

Figure 36 depicts a display screen showing information displayed in the parts editor window.

Figure 37 depicts a display screen showing information displayed in the parts editor window.

Figure 38 is a flow chart depicting procedures followed when a user deletes parts.

Figure 39 is a flow chart depicting procedures followed when a user moves parts.

Figure 40 depicts a display screen showing information displayed in the parts editor window.

Figure 41 shows the internal object representation for a class.

Figure 42 depicts a generic list.

Figure 43 illustrates the data structure for attribute data.

Figure 44 illustrates the data structure for an enumerator object.

Figure 45 illustrates the data structure for a unit family.

Figure 46 depicts the data structure for units.

Figure 47 depicts the data structures for a unit families.

Figure 48 shows the data structure for an enumerated derived unit.

Figure 49 depicts the data structure for an instance and associated parameters.

Figure 50 depicts the data structure for a parameter.

Figure 51 is an example of a schema with instances.

Figure 52 is a flow chart depicting how the handle manager responds to a request for the virtual memory address of an object

Figure 53 depicts the sequential layout of the dynamic file.

Figure 54 shows the general layout of the schema and instance files.

Figure 55 shows the layout of a file header.

Figure 56 shows the layout of a schema file object which represents a class in the knowledge base.

Figure 57 shows the layout of a schema file object which represents an attribute in the knowledge base.

Figure 58 shows the layout of a schema file object which represents an enumerator in the knowledge base.

Figure 59 shows the layout of a schema file object which represents a unit in the knowledge base.

Figure 60 shows the layout of a schema file object which represents a unit family in the knowledge base.

Figure 61 shows the layout of an instance file object.

Figure 62 shows the layout of a Type 1 dynamic object used to store a character string.

Figure 63 shows the layout of a Type 2 dynamic object used to store data items which are four bytes in length.

Figure 64 shows the layout of a Type 3 dynamic object used to store parameter data.

Figure 65 is a flow chart depicting how to add a class to the schema.

Figure 66 is a continuation of the flow chart in Figure 65.

Figure 67 is a flow chart depicting the addition of enumerated attributes.

Figure 68 is a continuation of the flow chart in Figure 67.

Figure 69 is a flow chart depicting the addition of an instance.

Figure 70 is a continuation of the flow chart in Figure 69.

Figure 71 is a flow chart depicting the deletion of a class.

Figure 72 is a continuation of the flow chart in Figure 71.
Figure 73 is a flow chart depicting the deletion of an attribute.
Figure 74 is a continuation of the flow chart in Figure 73.
Figure 75 is a flow chart depicting the deletion of an instance.
Figure 76 is a flow chart depicting the steps involved in moving a subtree.
Figure 77 is a continuation of the flow chart in Figure 76.
Figure 78 is a flow chart depicting unhooking a moved class from the original parent.
Figure 79 is a flow chart describing the process for finding the common ancestor of the class to be moved.
Figure 80 is a continuation of the flow chart in Figure 79.
Figure 81 is a graphical representation of the data maintained by the connection manager.
Figure 82 is a flow chart describing applying a local query.
Figure 83 is a continuation of the flow chart in Figure 82.
Figure 84 is a flow chart depicting the process for performing a query on a subtree.
Figure 85 is a flow chart depicting the application of a query count.
Figure 86 is a graphical representation of the locking function.
Figure 87 depicts match logic in genic.
Figure 88 depicts a display screen showing information displayed in the schema editor window.
Figure 89 depicts a display screen showing information displayed in the schema editor window.
Figure 90 is a flow chart depicting navigation of the class tree.
Figure 91 depicts a display screen showing information displayed in the schema editor window.
Figure 92 is a flow chart depicting reparenting a class to a new subclass.
Figure 93 depicts a display screen showing information displayed in the schema editor window.
Figure 94 depicts a display screen showing information displayed in the schema editor window.
Figure 95 is a flow chart depicting rearranging a class in the schema editor.
Figure 96 is the flow chart for the overall legacy procedures in the class manager.

Figure 97 depicts a display screen showing information displayed in the schema editor window.

Figure 98 depicts adding new classes in the schema editor window.

Figure 99 depicts a display screen showing information displayed in the schema editor window.

Figure 100 depicts a display screen showing information displayed in the schema editor window.

Figure 101 is a flow chart depicting rearranging attributes in the schema editor.

Figure 102 depicts a display screen showing information displayed in the schema editor window.

Figure 103 depicts a display screen showing information displayed in the schema editor window.

Figure 104 is a flow chart depicting the addition of a new enumerated attribute in the schema editor window.

Figure 105 depicts a display screen showing information displayed in the schema editor window.

Figure 106 is a flow chart depicting the addition of a numeric attribute.

Figure 107 depicts a display screen showing information displayed in the schema editor window.

Figure 108 depicts a display screen showing information displayed in the schema editor window.

Figure 109 is a flow chart depicting the addition of a Boolean attribute.

Figure 110 depicts a display screen showing information displayed in the schema editor window.

Figure 111 is a flow chart depicting the addition of a new string attribute.

Figure 112 depicts a display screen showing information displayed in the schema editor window.

Figure 113 is a flow chart depicting the addition and insertion of enumerators.

Figure 114 depicts a display screen showing information displayed in the schema editor window.

Figure 115 depicts a display screen showing information displayed in the schema editor window.

Figure 116 is a flow chart depicting the deletion of enumerator type attributes.

Figure 117 depicts a display screen showing information displayed in the schema editor window.

Figure 118 depicts the flow chart for editing a numeric attribute in the schema editor.

Figure 119 depicts a display screen showing information displayed in the schema editor window.

Figure 120 is a flow chart depicting the addition of values to a table.

Figure 121 is a picture of the automatic values dialog in the table editor in the schema editor.

Figure 122 is a flow chart of the process for adding labels in the table editor.

Figure 123 is a picture of the automatic labeling dialog in the table editor in the schema editor.

Figure 124 represents the process flow chart for the user changing the rows and columns of a table.

Figure 125 shows the command line parameters for import.

Figure 126 shows the command line parameters for simp.

Figure 127 is a flow chart for the user deleting an attribute in the schema editor.

Figure 128 is a picture of a screen in the schema editor with a class selected for attribute editing when the class has no locally defined attributes.

Figure 129 is another picture of a screen in the schema editor with a class selected for attribute editing when the class has attributes available for editing.

Figure 130 show the confirmation dialog that appears in the schema editor when deleting an attribute.

Figure 131 is an example of match criteria in genic.

Figure 132 is a flow chart depicting the process for legacy processing.

Figure 133 is a flow chart depicting automatic part classification function of the legacy manager.

Figure 134 is a flow chart depicting the method for classifying a part in the legacy process.

Figure 135 is a flow chart depicting legacizing ancestor parts.

Figure 136 is a flow chart depicting the method for legacizing an instance.

Figure 137 is a continuation of the flow chart in Figure 136.

Figure 138 is a continuation of the flow chart in Figure 137.

Figure 139 is a flow chart depicting processing the attributes for a class for classification.

Figure 140 is a flow chart depicting processing a thesaurus for a schema object.

Figure 141 is a flow chart depicting legacizing a class thesaurus.

Figure 142 is a flow chart depicting parameterizing a part instance.

Figure 143 is a flow chart depicting legacizing non-numeric attributes for a class.

Figure 144 is a diagram depicting the state of a query result before and after processing a sort request.

Figure 145 is a flowchart showing the legacy internal process for a numeric attribute.

Figure 146 is a flow chart depicting the internal working of the classifier.

Figure 147 is a continuation of the flow chart in **Figure 146**.

Figure 148 is a continuation of the flow chart in **Figure 147**.

Figure 149 is a continuation of the flow chart in **Figure 148**.

Figure 150 is a flow chart depicting the internal working of the schema generator.

Figure 151 is a continuation of **Figure 150**.

Figure 152 is a depiction of data structures in the database manager in the dynamic class manager.

Figure 153 is a flow chart of the internal processes of an import.

Figure 154 is a continuation of **Figure 153**.

Figure 155 depicts the data structures in the query manager after applying a query.

Figure 156 depicts the data structures in the query manager after setting a numeric query selector.

Figure 157 depicts the data structures in the query manager after setting a boolean query selector.

Figure 158 shows a numeric query selector class in the query manager.

Figure 159 shows an enumerated query selector class and a string query selector class in the query manager.

Figure 160 shows the base query or class and the boolean selector

class in the query manager.

Figure 161 depicts a query result class in the query manager.

Figure 162 depicts the base query class, the query class, and the search result class in the query manager.

Figure 163 represents the query manager class and the query handle manager class that are the main data structures in the query manager.

Figure 164 shows the classes that are created in the query manager after a query is created.

Figure 165 is a flow chart that depicts stages of processing in genic.

Figure 166 is a continuation of the flow chart of **Figure 165**.

Figure 167 is a continuation of the flow chart of **Figure 166**.

Figure 168 is a depiction of a typical server architecture for the invention.

Figure 169 is a depiction of a typical client architecture for the invention

Figure 170 depicts the process flow chart for the legacy knowledge base open screen.

Figure 171 shows the legacy knowledge base open dialog.

Figure 172 shows the screen that appears when the legacy application is invoked.

Figure 173 is the flow chart for the process after the legacy work area is selected.

Figure 174 is the main legacy screen.

Figure 175 depicts screen after the selection of a class for thesaurus editing.

Figure 176 is the flow chart of invoking a dialog for thesaurus editing.

Figure 177 show the thesaurus editing dialog for a class.

Figure 178 depicts the process flow for the thesaurus editing dialog in **Figure 177**.

Figure 179 shows the thesaurus editing dialog after adding a new entry.

Figure 180 shows text entered in a thesaurus entry.

Figure 181 shows a regular expression in a thesaurus entry.

Figure 182 shows the result of inserting a new thesaurus entry.

Figure 183 shows a complex regular expression in the thesaurus entry.

Figure 184 depicts the flow chart for invoking the thesaurus editor for an enumerated attribute.

Figure 185 depicts a display screen showing the procedure for bringing up a thesaurus editor for an enumerated attribute from the parts specification window.

Figure 186 depicts a display screen showing editing an enumerator thesaurus from the parts specification window.

Figure 187 depicts a display screen showing editing an enumerator thesaurus from the edit parts window.

Figure 188 is a diagram depicting the state of a query result before and after processing a request to retrieve an instance from a sorted query result.

Figure 189 depicts the management of sorted ranges within a sorted query result.

Figure 190 depicts a display screen showing the procedure of bring up a numeric attribute thesaurus editor from the edit parts window.

Figure 191 depicts a display screen showing the procedure for editing a numeric attribute thesaurus from the edit parts window.

Figure 192 depicts a display screen showing the procedure for editing a unit thesaurus.

Figure 193 depicts a flow chart for editing a unit thesaurus.

Figure 194 depicts a display screen showing the procedure for setting up legacy processing for selected parts.

Figure 195 depicts a flow chart for setting up legacy processing for selected parts.

Figure 196 depicts a display screen showing the result of legacizing selected parts.

Figure 197 depicts a flow chart for editing the list of attributes to parameterize.

Figure 198 depicts a display screen showing the procedure for editing a list of attributes to parameterize.

Figure 199 depicts a flow chart for generating customer schema from customer data.

Figure 200 depicts a flow chart for initially classifying customer data and generating an import map.

Figure 201 depicts a flow chart for augmenting customer data from a database of vendor parts.

Figure 202 depicts a flow chart for buffering query result to

optimize network performance.

Figure 203 depicts editing a non-enumerated thesaurus.

Figure 204 is a diagram of a network environment that is suitable for a preferred embodiment of the present invention.

Figure 205 is a block diagram depicting an overall architecture for a system employing a preferred embodiment of the present invention.

Figure 206A is a schematic diagram which depicts an extended database granularity hierarchy proposed in the past.

Figure 206B is a schematic diagram that depicts another example of a hierarchy of lock granules proposed in the past.

Figure 206C is a schematic diagram that depicts a hierarchy of lock granules in accordance with the present invention.

Figure 207A is a schematic diagram that depicts a hierarchy in which a class share lock has been applied to three classes.

Figure 207B is a schematic diagram that depicts a hierarchy in which a tree lock has been applied to a class, and in conjunction with Figure 207A, demonstrates an example of lock subsuming.

Figure 208 is a diagram representing lock conflicts for the lock types and granularities employed by the present invention.

Figure 209 is a diagram illustrating a hierarchy during a step in a process of granting a lock request.

Figure 210 is a diagram illustrating the hierarchy during a subsequent step in the process of granting a lock request.

Figure 211 is a diagram illustrating the hierarchy during a subsequent step in the
process of granting a tree lock request on a class whee the steps depicted in **Figure 209** and **Figure 210** are successful.

Figure 212 is a flow diagram representing the locking process performed when a retriever window is opened.

Figure 213 illustrates the process that occurs when a class is selected in the class hierarchy.

Figure 214 is a flow diagram that represents the process of opening a class to view subclasses.

Figure 215 is a flow diagram representing a process that occurs when a user selects a "find class" activity.

Figure 216 depicts an example of a screen display when navigating the schema by opening and selecting classes.

Figure 217 is a diagram of a schema illustrating an example of

internal lock states of classes in the schema corresponding to the display of **Figure 216.**

Figure 218 illustrates a lock table maintained by the lock manager as correlated with the schema depicted in **Figure 217.**

Figure 219 is a diagram that illustrates the contents of one of the lock objects in the lock table shown in **Figure 218.**

Figure 220 diagrams the process that occurs when a user adds a part to a class in the knowledge base.

Figure 221 shows a schema having a class to which a part is being added.

Figure 222 depicts the lock table states for the process of adding a part as described in **Figure 220.**

Figure 223 shows a lock object corresponding to the class for the add part operation corresponding to **Figures 221-222.**

Figure 224 depicts an example of a screen display when adding a part to the schema.

Figure 225 illustrates a flow chart for an example where a user has selected the edit parts function.

Figure 226 illustrates a flow chart for an example where a user, while in the edit parts window, navigates to different locations in the class hierarchy tree.

Figure 227 depicts an example of a screen display when editing a part.

Figure 228 shows a schema corresponding to the schema being edited in **Figure 227.**

Figure 229 shows a lock holder table after completion of the creation of an edit parts window.

Figure 230 shows a lock object corresponding to the example shown in **Figures 227-229.**

Figure 231 shows a flow chart for an example of moving a single part from one class in a subtree to another class within a given subtree.

Figure 232 shows a flow chart for an example of a general case of moving any number of parts from one class in a subtree to another class within that subtree.

Figure 233 shows a lock holder table during the process for the general case of moving any number of parts from one class in a subtree to another class within that subtree.

Figure 234 shows details of the lock objects for the source and

destination classes, and the associated actions for the general case of moving parts shown in Figure 232.

Figure 235 shows a preferred display associated with a move parts operation.

Figure 236 is a flow chart illustrating the process for an optimized case where one part is to be removed from the knowledge base.

Figure 237 is a flow chart illustrating the process for a general case of deleting one or more parts from a subtree.

Figure 238 shows the locks that must be held by a lock holder that wishes to remove an instance from a class.

Figures 239 and **240** show preferred displays associated with a delete parts operation.

Figure 241 is a flow chart that describes steps that are involved in concurrency control when using the schema editor to change the structure of the schema.

Figure 242 shows a lock table that indicates the locks that are held during the operations described in Figure 241.

Figure 243 illustrates a screen display for a preferred embodiment showing a schema developer window that is opened in one step of the process shown in **Figure 241.**

Figure 244 shows a flow chart illustrating the mechanisms that are used by the concurrency control means when displaying a instance.

Figure 245 depicts the lock table, a diagram of the schema, and details concerning one of the lock obje cts, showing the condition of the lock holder table for the situation depicted in **Figure 244.**

Figure 246 illustrates a screen display for a preferred embodiment showing a search results window that is opened in one step of the process shown in Figure 244.

Figure 247 is a flow chart depicting the steps for requesting authorization to do a schema edit.

Figure 248 is a flow chart depicting the steps for requesting authorization to do an instance edit.

Figure 249 is a flow chart depicting the steps for requesting a class share lock.

Figure 250 is a flow chart depicting the steps for requesting a tree share lock.

Figure 251 is a flow chart depicting the steps for requesting a tree update lock.

Figure 252 is a flow chart depicting the steps for requesting a tree exclusive lock.

Figure 253 is a chart representing the application of a lock manager by a knowledge base client.

Figure 254 is a diagram of a lock table that is used by the lock manager.

Figure 255 shows the data structure for the lock holder table.

Figure 256 is a flow chart showing the operation of starting a lock holder.

Figure 257 is a flow chart for the operation of ending a lock holder.

Figure 258 shows the major components of a computer hardware configuration for a knowledge base server.

Figure 259 shows the major components of a computer hardware configuration for a retriever, a schema editor, a graphical user interface component, and an API.

Figure 260 and **Figure 261** depict flow charts for the process of comparing part attributes.

Figure 262A shows an example of a display of a search results window.

Figure 262B shows an example of a display of a compare parts dialog box.

Figure 263 shows an example of a display of a compare parts dialog box after a compare to selected part command has been invoked.

Figure 264 depicts an initial display screen showing the part specification window.

Figure 265 depicts an example of the part specification window during a search.

Figure 266 depicts a display screen showing information displayed in the part specification window.

Figure 267 is a flow chart depicting procedures followed when a user edits parts.

Figure 268 depicts a display screen showing information displayed in the parts editor window.

Figure 269 depicts a display screen showing information displayed in the parts editor window.

Figure 270 is a flow chart depicting procedures followed when a user deletes parts.

Figure 271 is a flow chart depicting procedures followed when a user moves parts.

Figure 272 depicts a display screen showing information displayed in the parts editor window.

Figure 273 shows the internal object representation for a class.

Figure 274 depicts a generic list.

Figure 275 illustrates the data structure for attribute data.

Figure 276 illustrates the data structure for an enumerator object.

Figure 277 depicts the data structure for an instance and associated parameters.

Figure 278 depicts the data structure for a parameter.

Figure 279 is an example of a schema with instances.

Figure 280 is a flow chart depicting how to add a class to the schema.

Figure 281 is a continuation of the flow chart in Figure 280.

Figure 282 is a flow chart depicting the addition of enumerated attributes.

Figure 283 is a continuation of the flow chart in Figure 282.

Figure 284 is a flow chart depicting the addition of an instance.

Figure 285 is a continuation of the flow chart in Figure 284.

Figure 286 is a flow chart depicting the deletion of a class.

Figure 287 is a continuation of the flow chart in Figure 286.

Figure 288 is a flow chart depicting the deletion of an attribute.

Figure 289 is a continuation of the flow chart in **Figure 288.**

Figure 290 is a flow chart depicting the deletion of an instance.

Figure 291 is a flow chart depicting the steps involved in moving a subtree.

Figure 292 is a continuation of the flow chart in **Figure 291.**

Figure 293 is a flow chart depicting unhooking a moved class from the original parent.

Figure 294 is a flow chart describing the process for finding the common ancestor of the class to be moved.

Figure 295 is a continuation of the flow chart in **Figure 294.**

DETAILED DESCRIPTION OF A PREFERRED EMBODIMENT

The present invention can advantageously be used in a network environment. A number of configurations are possible, and only one example will be described herein. It should be understood that the present description is illustrative, and the invention is not limited to the particular example or configuration described herein. An overview of a suitable network environment is depicted in **Figure 2**.

The network 100 includes a first UNIX server host 101. One or more knowledge bases 123are installed on the first UNIX server host 101. In the illustrated example, a first knowledge base server daemon 102 runs on the first UNIX server host 101. Data may be physically stored on a first disk drive 103 which is sometimes referred to as secondary storage. More than one knowledge base server 102 may exist on the system 100. For example, a second knowledge base server daemon 104 may be provided. Similarly, data may be physically stored on a second disk drive 105. The first UNIX server host 101 may communicate over a network with a second UNIX server host 106 and a third UNIX server host 107. In this example, a registry server daemon 108 is installed on the second UNIX server host 106. The registry server daemon 108 could run on the same UNIX server host 101 as the knowledge base server daemons 102 and 104. Certain files containing information used by the registry server 108 may be physically stored on a third disk drive 109. The registry server 108 is used to administer user access to features and access to knowledge bases. The registry server 108 also allows a system administrator to set up different user profiles for different types of users. For example, there may be some users who only need access permission to retrieve parts from a knowledge base 123. Other users may need access permission to add parts, or edit existing parts. The registry server 108 provides a convenient way to specify and control user access to specific functions. The registry server 108 describes the knowledge bases in use, the users that are allowed to use the system, and the access rights that each user has to the listed knowledge bases.

A licensed manager server daemon 110 is installed on the third UNIX server host 107. The license manager server 110 controls the number of licenses available to any authorized user on the

network 100. The license manager 110 uses "floating" licenses. For example, when 20 licenses are available through the license manager 110, any 20 users of the network can use these licenses concurrently.

Before a knowledge base server 102 can be started, the license manager server 110 and the registry server 108 must be running. In order for the registry server daemon 108 to continue to run, it must be able to obtain a license from the license manager server 110. If the registry server 108 can not contact the license manager server 110, it will exit. Therefore, the license manager server 110 should be started first. The registry server 108 should be started second. The knowledge base server 102 should be started thereafter.

Users may access data available through the knowledge base server daemon 102 or the knowledge base server daemon 104 using a suitable workstation 111 connected to the network 100. For example, a Sun Microsystems SPARCstation 111, preferably running X11R5/Motif v1.2 software. Alternatively, a SPARC compatible workstation may be used. In addition, a Hewlett Packard series 700 workstation running Motif v1.2 with X11R5 will also give satisfactory results. In the illustrated example, the Sun Microsystems SPARCstation 111 runs a SunOS 4.1.x operating system. A Hewlett Packard series 700 platform preferably includes HP-UX 9.0.x software.

In addition, a user can access the network 100 using an IBM PC compatible computer 112 running Microsoft Windows v3.1. In the illustrated example, the IBM PC compatible computer 112 can be a 386, 486, or Pentium based machine. The IBM PC compatible computer 112 includes a display 113, a mouse 114, and a keyboard 115. The display 113 is preferably a VGA or SVGA CRT 113. In the illustrated example, the IBM PC compatible computer 112 runs MS-DOS 5.0 or later disk operating system, MS-DOS 6.2 being preferred. The IBM PC compatible computer 112 also must have Winsock 1.1 compliant TCP/IP software. A windows client using an IBM PC compatible computer 112 will employ RPC calls via TCP/IP to communicate with the knowledge base server 102. The IBM PC compatible computer 112 should have sufficient available disk space for software installation. In the illustrated example, the IBM PC compatible computer 112 should also have at

least 4 megabytes of RAM memory; 16 megabytes of memory is preferred.

The Sun Microsystems SPARCstation 111 similarly has a display 116, a mouse 117, and a keyboard 122.

The illustrated network 100 shown in **Figure 2** also supports an X Windows client which employs a computer 118, which has a display 119, a mouse 120, and a keyboard 121. A user can access this system using X Windows in a proper emulation mode interfacing with the workstation 111.

In the example shown in **Figure 2**, each of the server hosts 101, 106 and 107 may be a Sun Microsystems SPARCstation (or a SPARC compatible), or a Hewlett Packard series 700 computer. In a presently preferred embodiment, a single UNIX system on the network may be designated to run the knowledge base server daemon 102, the registry server daemon 108, and the license manager server daemon 110. This implementation may provide ease of administration. For best performance, the software and knowledge bases embodying the present invention should reside on a single server host 101 local disk drive 103. However, a knowledge base 123 for example may reside on a remote disk drive 109.

In the present example, the network environment includes an operating system with a file system, supports virtual memory, employs UDP/TCP/IP protocol, and provides ONC/RPC (open network computing/remote procedure call) services. In addition, it is useful if the network environment supports multiprocessing and multitasking.

The present system supports interactive editing by the user. Users are able to change the parts schema by adding and deleting part attributes, and are able to add whole sections to the schema to support their custom parts. In addition to schema editing, parts in the database may be repositioned within the schema hierarchy, as well as being modified, added, and deleted.

The present invention provides an object oriented tool set that (1) supports dynamic class management, (2) supports a database having a large number of parts (e.g., in excess of several hundred thousand parts), (3) has performance sufficient to support interactive retrieval of parts by hundreds of users, and (4) understands and automatically manages the translation across different units of measure. This system may be referred

to as a knowledge base management system.

The present knowledge base management system enables a user to locate "objects" by describing them in terms of their attributes. In a parts management application, these objects are parts, but it could be any other item described by a collection of attributes. Applications are created by subject matter experts--not computer programmers. The sophistication of the application is tied to the development of the subject based schema, not to computer program development.

The present invention may be better understood in connection with the following description of the overall architecture of a presently preferred embodiment.

I. Overall Architecture

Turning now to **Figure 3**, presently preferred embodiment may include a retriever 130, a knowledge base client 131, and a knowledge base server 132. A legacy manager 133 is preferably included to facilitate organization of an existing legacy database into a schema for use in connection with the present invention. In a preferred embodiment, the knowledge base server 132 includes a dynamic class manager 134, a connection manager 135, a query manager 136, a handle manager 137, a units manager 138, a database manager 139, and a file manager 140. A preferred system also includes a registry server 141 and license manager 142 to control unauthorized user access to the system.

A schema editor 144 is preferably provided to modify or customize the schema. An application programming interface or API 143 is also provided in the illustrated environment.

A knowledge base 123is a database containing information, and is stored on a disk drive 103. The knowledge base 123 in the present example comprises three files: the schema file, the variable data file, and the instance file. A schema is a collection of classes, attributes, units, and unit families and their relationships.

In the present example, the executable for the knowledge base server 132 is **pmxdbd**. Each **pmxdbd** server provides access to one knowledge base 123. Therefore, the UNIX server host 101 must run one **pmxdbd** process for each knowledge base 123. For example, in a system having three knowledge bases, the UNIX **ps** command would show three **pmxdbd** servers running.

Unlike an RDBMS based application, with the present knowledge base management system solution, complexity, and thus response time, does not increase exponentially with size and number of relationships. Knowledge is not tied to the quantity of software code. Schema can be dynamically updated without recompiling the application. Data and schema are interactively user modifiable. A query is equivalent to finding corresponding indices, not computing RDBMS table joins. Database size is reduced. A knowledge base management system database 123 in accordance with the present invention is typically about 1/10 the size of an equivalent RDBMS database.

The steps for logging into the system are shown in the flowcharts depicted in **Figure 4A** and **Figure 4B**.

A login procedure is initiated by a user logging into the retriever 130, as depicted in step 150 in **Figure 4A**. The user's name and password are sent to the registry server 141, as shown in 151. In step 152, the user name and password are validated by the registry server 141. If the user name and password are not valid, the flow returns to step 150 and the user must try again. If the name and password are valid, the flow continues to step 153 in which the retriever 130 asks for an appropriate software license from the license manager 142.

In step 154, the license manager 142 determines whether or not a license is available for the user. If a license is not available, flow returns to step 150 shown in **Figure 4A**. If the license is available, license manager 142 grants a license to run and flow continues to step 155. The retriever 130 will display on the display 116 a list of knowledge bases 123 which are available. The list of knowledge bases is obtained from the registry server 141. The registry server 141 will only return a list of knowledge bases for which the user has access rights. In step 156, the user may then select a knowledge base 123 to open.

In step 157, the retriever 130 will send an open knowledge base request to the knowledge base server 132. In step 158, the knowledge base server checks to see if the requested knowledge base 123 is locked. There are times, for example when an input administrator is performing extensive input into a knowledge base 123, when it is desirable to lock a knowledge base 123 and

temporarily prevent any other user from accessing it. Instances when a knowledge base 123 is locked are typically those in which one person needs to have exclusive access to the knowledge base 123. If the knowledge base 123 is locked, flow returns to step 155 in which the retriever 130 again displays on the display 116 a list of knowledge bases from the registry server 141 for which the user has access rights. The user will also receive a message notifying the user that the knowledge base 123 that the user initially attempted to open is locked.

If the requested knowledge base 123 is not locked, flow continues to step 159 in **Figure 4B**, and the knowledge base server 132 checks to determine which open modes are valid for this user or knowledge base 123. For example, if the knowledge base 123 is read only, and the user has attempted to access it in a mode in which a write operation to the knowledge base 123 has been requested, flow returns to step 155 in **Figure 4A** and the user receives a message on the display 116.

In this example, if the requested open mode is available that particular knowledge base 123 for that particular user, flow continues to step 160 shown in **Figure 4B**. The knowledge base server 132 attempts to get the appropriate software license from the license manager 142. If a license is not granted, flow returns to step 155 shown in **Figure 4A.** If a license is available, flow continues to step 161 shown in **Figure 4B**. In that event, the knowledge base server 132 will return connection and a knowledge base handle to the retriever 130. The user will then have successfully logged on to the network 100 and will have access to the requested knowledge base server 102.

Figure 168 shows the major components of a computer hardware configuration 101 providing the computational and communications environment for a knowledge base server 132. This configuration consists of a central processing unit or CPU 2109 which includes an arithmetic logical unit 2100 which fetches and executes program instructions from main memory 2101. The programs are stored on a disk drive 103, access to which is provided through a disk controller 2106. The knowledge base files 123 are also stored on disk drive 103 and accessed through virtual memory addresses 2112 in main memory 2101, through which, when required, a page 2111 of contiguous data in a disk file 2108 is copied into

main memory 2101. The preferred embodiment of the present invention uses virtual memory 2112 for this knowledge base management system. The knowledge base server 132 interacts with the client API 143 through a local area network 100, access to which is controlled by network controller 2102, or through a wide area network 2104, access to which is controlled by a serial interface controller 2103. An I/O bus 2105 mediates data transfers between the CPU 2109 and the peripheral data storage, interface and communication components.

Figure 169 shows the major components of a computer hardware configuration 112 providing the computational and communications environment for a retriever 130, schema editor 144, a graphical user interface component of legacy 133, and an API 143. This configuration consists of a central processing unit or CPU 2109 which includes an arithmetic logical unit 2100 which fetches and executes program instructions from main memory 2101. The programs are stored on one or more disk drives 2110, access to which is provided through a disk controller 2106. The user interacts with the system through the keyboard 115 and mouse or similar graphical pointer 114 with the graphical user interface displayed on the CRT display 113. The API 143 communicates with the knowledge base server 132 through a local area network 100, access to which is controlled by network controller 2102, or through a wide area network 2104, access to which is controlled by a serial interface controller 2103. An I/O bus 2105 mediates data transfers between the CPU 2109 and the peripheral data storage, interface and communication components.

A. Retriever

The retriever 130 is an application that provides a graphical interface for finding and managing parts. The retriever 130 communicates with the knowledge base client 131 using the API 143. The retriever 130 provides an object oriented graphical user interface. A user interacts with the retriever 130 providing input through a keyboard 115 and a mouse 114. The retriever displays information on the display 116.

Figure 5 depicts a typical display that appears on the screen of the display 116 after a user successfully logs on to the system. The particular example described herein is described in a Windows environment, it being understood that the invention

is not limited to implementation in Windows. Those skilled in the art are familiar with windows techniques and instructions, including how to click, double click, drag, point and select with a mouse 114. Additional information may be obtained from the Microsoft Window's User's Guide (1992), available from Microsoft Corporation, One Microsoft Way, Redmond, Washington, 98052-6399, part number 21669.

When a user first opens a knowledge base 123, a part specification window 170 appears, as shown in **Figure 5**. Initially, the left hand portion of the screen 171 displays the parts found 172, which in this instance is the total number of parts found in the knowledge base 123. Also displayed on the left-hand portion of the screen 171 is the root class 173 and the root subclasses 174. In the illustrated examples, the root subclasses 174 are electrical components, mechanical (i.e., mechanical components), and materials. The root class 173 is the upper most class that has no parent. In this example, it is the name of the knowledge base 123, or the very beginning of the schema. A subclass 174 is a class that has a parent. When a class is chosen, any subclasses that belong to that class will appear on the display 171. Subclasses are the children of the parents. For example, the parent of the mechanical subclass 174 is the root class 173, and the mechanical subclass 174 is a child of the parent root class 173. In the example shown in **Figure 5**, there are three subclasses.

The right hand portion of the screen 175 displays root attributes 176. In the illustrated example, the attributes are part number, description, and cost. Attributes 176 are the characteristics of a class or subclass 174.

Specific attribute values may be entered to locate a part as search criteria 177. Command buttons 178 are displayed in the part specification window 170. When a command name is dimmed, the command is not available at the current time. In the example shown in **Figure 5**, display button 179 will, when activated, display a list of the parts matching the current specification at that point in the search. An edit button 180 and a make button 181 are also shown in **Figure 5**. These command buttons are only shown if the user is authorized to edit attribute values and has access rights to make a new part, respectively. When

activated, the edit button 180 causes a parts editor window to be displayed which allows the user to edit attribute values. When the make button 181 is activated, it allows a user to add a part to the knowledge base 123. These three buttons appear in the parts area 186 of the command buttons 178. A display order area 187 has a set all command button 182 and a clear command button 183. When the set all button 182 is activated, it sends a sequential display order to each attribute. This order is used to arrange the attributes display in a search results window. When the clear button 183 is activated, it causes the display order numbers to be removed from all of the attributes.

A clear criteria area 188 includes an all button 184 and a selected button 185. When the all button 184 is activated, it causes all values to be cleared from the search criteria fields 177. When the selected button 185 is activated, it causes the value to be cleared from the selected search criteria fields 177.

The left-hand portion of the screen 171 is separated from the right-hand portion of the screen 175 by a split bar 189. A user may drag the split bar 189 to the left or right to change the size of the left hand portion of the screen 171 and the right-hand portion of the screen 175.

Icons are displayed in the part specification window 170 to provide information to the user. Closed folder icons 189 are used to represent the classes that have subclasses. An open folder icon 190 is used to represent opened classes 173 and subclasses. A protected icon 191 indicates an attribute for which the user cannot enter values when making a part. An undefined icon 192 indicates a column which, if selected by a user, will be used to search for parts that do not have any value for the selected attribute 176. Also shown in **Figure 5** is a text icon 193 associated with each of the attributes entitled "part number", "description", and "cost". The text icon 193 is used to indicate attributes 176 that have values consisting of a string of characters. For example, a written description is an attribute 176 of a part that is a text attribute. An order column 194 is used to indicate the sequence, from left to right, in which attributes 176 will appear when a search results window 299 is displayed. When applicable, a "1" in this column will indicate the attribute 176 that will be displayed on the far left

of the search results window, a "2" in this column will indicate the attribute 176 that is displayed next, and so forth from left to right.

The part specification window 170 also contains a query type indicator 195. This only appears for users who have access rights for editing parts. This indicates the type of query that the user is performing, i.e., whether it is global or local. In the example illustrated in **Figure 5**, the query type has defaulted to global.

When the user needs to locate a part, the user generally knows the characteristics or attributes 176 of the part, but the user may not know the part number. By knowing the attributes 176, the user can easily locate the part in the knowledge base. A user locates parts in the knowledge base by specifying the type of part the user wants to find. The user specifies a part by selecting the part's class 173 and subclasses 174 and by entering attribute search criteria 177.

The first step in specifying the part the user wants to locate is to open the class 173 the part belongs to. When the user opens a class such as the mechanical class 174 shown in **Figure 5**, the user sees the next level of the hierarchy, i.e., subclasses 196 shown in **Figure 6**. The closed folder icon 189 next to the mechanical subclass 174 shown in **Figure 5** is replaced by an open folder icon 190 shown in **Figure 6**. The next step in specifying a part is to open the next subclass 196 the part belongs to, in this example the fasteners subclass 196. When the user opens one of the subclass folders 196, the user sees the next level of subclasses 197, as shown in **Figure 6**. The user continues specifying a part by opening another level of subclass 197, such as the bolts subclass 197 shown in **Figure 6**, and so forth through lower levels of subclasses 198 and 199, until the user reaches a class that has no more subclasses, which is called a leaf class 201. A leaf class is identified with a page icon 202. An open class 199 will be displayed with a line 232 forming a subtree connecting the subclasses 204 of the class 199. More specifically, line 232 connects the open folder icon 190 for the class 199 with the closed folder icons 189 of the subclasses 204. The line 232 extends vertically down from the open folder icon 198 to the level of the last subclass 204, as horizontal branches

connecting the vertical line 232 with closed folder icons 189 for the subclasses 204.

At every subclass level the number of parts found at that level is displayed as parts found 172. The parts found number 172 indicates the number of parts located within the current subclass 199, including its subclasses and leaf classes (see **Figure 6**). This instant feedback to the user greatly facilitates a search.

The steps followed by the retriever 130 are depicted in **Figure 7**. The user selects a class in step 205. In step 206, the selected class is displayed in a highlighted representation 200 (see **Figure 5**). The retriever 130 resets the current query to the selected class in step 206.

Referring to **Figure 7**, in step 207 of the flow the retriever 130 determines whether inherited attributes 176 have query selectors 177 set. If query selectors are set, the flow proceeds to step 208, and the retriever 130 sets corresponding query selectors for inherited attributes 176. Flow then proceeds to step 209. In step 207, if inherited attributes do not have any query selectors set, the flow proceeds directly to step 209.

In step 209, the retriever 130 gets local attributes 203 for the class 199 and adds them to the display in the right hand portion 175 of the part specification window 170, which may be referred to as the attribute window 175.

Flow proceeds to step 210 where the retriever 130 updates the part count and displays that information as parts found 172. Flow then proceeds to step 211 where control is returned to the user and the system waits for another command.

The procedure 210 for updating the part count and display is shown in more detail in **Figure 8**. In order to update the part count and display, the retriever 130 gets a value representative of the query result count and displays it as parts found 172. This is shown in step 212 of **Figure 8**.

The retriever 130 then checks to determine whether the part count is zero, which is performed in step 213. If the part count is zero, the retriever 130 checks to determine whether this particular user has access rights to add a new part to the knowledge base. This occurs in step 214. If the user does not have such access rights, the flow proceeds to step 211 and

returns control to the user. If the user does have such access rights, the retriever 130 then activates the make button 181. Up until this point in time, the make button 181 had been dimmed because that procedure was not available. In a preferred embodiment, a user is not allowed to add a new part to the knowledge base unless the user has access rights which permit him or her to do so. Activation of the make button occurs in step 215 of the flow chart illustrated in **Figure 8**.

If the part count is not equal to zero, flow transfers to step 216 in which the retriever 130 checks to determine whether the user has access rights to edit parts. If the user does have such access rights, the retriever 130 proceeds to step 217 in the flow and activates the edit button 180. The flow then proceeds to step 218, where the retriever 130 activates the display button 179. In step 216, if the user does not have access rights to edit parts, the flow proceeds directly to step 218. After the display button is activated in step 218, the flow proceeds to step 211 where control is returned to the user.

Figure 9 depicts steps performed by the retriever 130 to open a class. In order to open a class such as the fasteners class 196 shown in **Figure 6**, the user positions the cursor to point to the closed folder 189 immediately next to the class and double clicks. As shown in **Figure 9**, the retriever 130 then changes the display of a closed folder 189 and replaces it with an open folder icon 190 in step 220. The retriever gets a list of the subclasses 197.

In step 221, the retriever 130 proceeds through the list of subclasses and determines whether the next subclass in the list is a leaf class 201. If it is, flow proceeds to step 222 in **Figure 9** and a page icon 202 is displayed for that subclass 201. Control will then proceed to step 224.

In step 221, if the next subclass in the list is not a leaf class, flow proceeds to step 223 where the retriever 130 displays a closed folder icon 189 and the class name for the subclass 197. Flow then proceeds to step 224.

In step 224, the retriever 130 checks to determine whether there are any more subclasses 197 in the list to display. If there are, flow proceeds back to step 221. If there are not, flow proceeds to step 205.

flow proceeds to step 205.

The procedure followed by the retriever 130 to close and open class 199 (see **Figure 10**) is depicted in **Figure 11**. In step 225, the user double clicks on the open folder icon 190 associated with the class 199 that is to be closed. Flow then proceeds to step 226 in **Figure 11**. In step 226, the retriever 130 removes all lines for the subtree 232 from the display 171. The names of the subclasses 204 and the closed folder icons 189 associated with them will also be removed from the display 171. The representation of the tree structure will then be collapsed to eliminate the space formerly occupied by the subclasses 204.

The flow then proceeds to step 227, in which the open folder 190 next to the parent class 199 is replaced with a closed folder icon 189. The flow then proceeds to step 228 in **Figure 11**, and control is returned to the user.

The changes to the part specification window 170 which occur when a user closes an open class may be better appreciated by comparing **Figure 10** with **Figure 6**. Starting with **Figure 10**, if the user closes the numeric class 199, the subtree line 232 and the subclasses 204 will be removed from the display, and the open folder icon 190 associated with the numeric class 199 will be replaced with a closed folder icon 189. The display 171 will be updated to appear as depicted in **Figure 6**.

Attributes 203 that may have a value taken from a set of pre-defined values are referred to as enumerated attributes. For example, the head style attribute 203 under bolts is taken from a set of pre-defined values because the head style for a bolt can only be one type taken from a finite list of possible head styles. Enumerated attributes such as head style 203 are identified with an associated enumerated icon 233.

Attributes 203 that have values of either true or false are Boolean attributes. For example, a bolt can have an attribute 203 used to indicate whether or not it is a self locking bolt. If the bolt is self locking, this attribute value is true. If the bolt is not self locking, this attribute value is false. Boolean attributes have a Boolean attribute icon 234 associated with them.

Attributes 236 that have values that are numeric and have an associated unit of measurement are called numeric attributes.

For example, the length of a bolt 236 is a numeric attribute. Numeric attributes 236 have a numeric attribute icon 235 associated with them.

The user may further specify the part by entering attribute search criteria 177. Each class and subclass has an associated set of attributes 176. Attributes 176 are the characteristics of a part, such as the material the part is made from, its length, or finish. As the user opens additional subclasses 196, 197, 198 and 199, the attributes 203 specifically associated with those subclasses 196, 197, 198 and 199 appear. These attributes 203 are the local attributes of the open class/subclass 199 and are appended to the existing attributes 176, which are the inherited attributes. By entering attribute search criteria 177, the user can narrow down the number of parts the user needs to check for applicability.

Referring to **Figure 12**, there may be many parts in the selected subclass or leaf class 240, and it is desirable to further specify a part by entering search criteria 177. When a user enters search criteria 177, the retriever 130 immediately performs a search to locate only the parts that have all of the specified attribute values 176, 203. There are four types of attributes: (1) enumerated, (2) numeric, (3) text, and (4) Boolean. The retriever 130 has different procedures for entering and clearing the search criteria for each type of attribute.

A procedure for entering text attribute search criteria 177 is shown in **Figure 12**. In step 250, the user selects the text attribute. For example, a user could enter search criteria 242 for the part number attribute 241. To do so, the user would click on the text attribute icon 193 associated with the part number attribute 241. The retriever 130 then pops up a text search criteria dialog box 237, as shown in **Figure 13**. The retriever 130 positions the cursor at the left most position the data entry field 243 of the text search criteria dialog box 237. Referring to **Figure 12**, the retriever 130 then proceeds to step 251 in order to accept text input entered in the data entry field 243.

The retriever 130 allows for the use of special characters as part of the search criteria 242. An asterisk matches any number of characters. For example, it may be desirable to locate

all parts containing the abbreviation "pf" (for picofarads) anywhere in the selected part number text attribute 241. To accomplish this, a user could type *pf* in the text data entry field 243. In the example illustrated in **Figure 13**, typing 015* in the data entry field 243 will cause the retriever 130 to search for any part number beginning with the digits 015 regardless of how many additional characters or numbers follow in the part number. A question mark matches any single character. For example, to locate all parts with descriptions containing fractional sizes, i.e., 1/8, 1/4, 1/2, etc., a user could type ?/?. Finally, occasions may arise when it is desirable to search for a part containing the special character * or the special character ? in the text. By typing | immediately before either special character * or ?, the retriever 130 will recognize the following special character as a regular character.

The text attribute search criteria 242 is case insensitive. A search will match a character regardless of whether it is upper case or lower case. The case of the letters typed by the user in the text data entry field 243 is disregarded when the search is looking for matching attribute values.

The user remains in control until user's input in the text data entry field 243 is confirmed. User may confirm such input by clicking on an OK button 244 in the text search criteria dialog box 237, or by pressing the enter key on the keyboard 115.

This may be better understood in connection with the following discussion of an example of the classes and subclasses a user may open to specify and locate a particular bolt.

A cancel button 245 is provided in the text search criteria dialog box 237 to enable a user to abort the text search. If the cancel button 245 is activated, the retriever 130 returns to the state that existed just prior to the user's selection of the associated text attribute icon 193. Any input entered in the text data entry field 243 will be ignored.

A clear button 246 is provided in the text search criteria dialog box 237. If this button is activated, the retriever 130 will clear any entry in the text data entry field 243. The dialog box 237 will remain in position, and the retriever 130 will continue to wait for the confirmation of input from the

user.

If the user enters characters in the text data entry field 243 and activates the OK button 244, flow continues to step 253 shown in **Figure 12**. The retriever 130 will add a text selector to the current query if none currently existed for the associated attribute 241. If a pre-existing text selector is present, the retriever 130 will replace it in the current query. The search will then be performed including this text attribute search criteria 242, and the retriever 130 will proceed to step 210 shown in **Figure 12**.

The text data entered by the user in the entry field 243 will be displayed in the search criteria field 242 in the right hand portion 175 of the part specification window 170. The parts count 172 will be updated as shown in **Figure 13** to reflect the results of the search.

If the length of the text data that is entered by the user exceeds the size of the data entry field 243, scroll buttons 247 may be used in the manner known to those skilled in the art to view text that may fall outside the field of view 243.

Numeric attributes such as the length attribute 236 shown in **Figure 13** may be selected as a search criteria 177. Referring to **Figure 14**, a user selects a numeric attribute such as the numeric attribute length 236 shown in **Figure 13** by clicking on its associated numeric attribute icon 235. Referring to **Figure 14**, this is represented by step 255 in the flow chart.

The retriever 130 then proceeds to step 256 to determine if a table of standard values has been defined for the selected numeric attribute 236. If no table of standard values has been defined, the retriever 130 proceeds to step 257. A custom numeric dialog box 265 shown in **Figure 15** will appear. Custom numeric dialog box 265 allows entry of a range of numeric values. A "from" numeric input field 266 is provided in the custom numeric dialog box 265. A "to" numeric input field 267 is also provided in the custom numeric dialog box 265. When a user types the "from" value, it is automatically copied to the "to" value field 267. If a user wants to search for only one value using the default unit of measure 268, the user may confirm the input having the same value in both the "from" input field 266 and the "to" input field 267 by clicking the OK command button 270, thus

proceeding to step 260 shown in **Figure 14**.

In step 257, a user may select a different unit of measure other than the default unit of measure 268. The default unit of measure is displayed in the custom numeric dialog box together with a button 269 which, when actuated, produces a drop down list box containing a list of other available units of measure. Thus, a different unit of measure may be selected from the drop down list box.

The custom numeric dialog box 265 includes a cancel button 271 and a clear button 272 which operate in a manner known to those skilled in the art. The custom numeric dialog box 265 also includes an OK button 270 described previously. A user's input is confirmed in step 260 shown in **Figure 14** when the user clicks the OK button 270 or presses the enter key on the keyboard 115 when the OK button 270 is highlighted.

Referring to **Figure 14**, in step 256 if a table of standard values is defined for the numeric attribute 236, the user is presented with a table of standard values 273 (see **Figure 16**) in step 258. Referring to **Figure 16**, if the list of standard values that are defined for the numeric attribute 236 exceeds the number that can be displayed in the pop up window 273 for the table of standard values, scroll buttons 274 may be used in a manner known to those skilled in the art to view values which are not currently displayed in the table of standard values window 273. A plurality of standard values 275 which are defined for the numeric attribute 236 are displayed in the table of standard values window 273, as shown in **Figure 16**. The currently selected standard value 276 is displayed in a highlighted manner.

The table of standard values window 273 includes an OK button 277. The retriever 130 will accept the highlighted standard value 276 when the user actuates the OK button 277. The table of standard values window 273 also includes a cancel button 278 and a clear button 280 which operate in a manner known to those skilled in the art.

The table of standard values window 273 includes a custom button 279. If the user does not find a standard value 275 on the list which is the same as the numeric value that is required, the user may actuate the custom button 279. In **Figure 14**, the retriever 130 checks for actuation of the custom button in step

259. If the custom button is actuated, flow proceeds from step 259 to step 257. The user is then presented with the custom numeric dialog box 265 shown in **Figure 15**. The user may then enter the desired numeric value in the manner previously described with reference to step 257 in **Figure 14**.

In step 260, when the user confirms the input by actuating the OK button 277 or by pressing the enter key on the keyboard 115, flow proceeds from step 260 to step 261. The retriever 130 will add or replace the numeric selector in the current query, depending upon whether the numeric attribute 236 was present in the previous query. The retriever 130 will then proceed to step 210 and update the parts count 172 and update the display 170. The table of standard values window 273 will disappear when the user's input is confirmed, for example, by actuating the OK button 277. In step 262, the retriever 130 then returns control to the user and awaits another command. The selected numeric value will be displayed in the numeric search criteria field 281.

The procedure for entering search criteria 282 for a Boolean attribute 203 shown in **Figure 17**. The user selects a Boolean attribute 203 that uses a search criteria 177 by clicking on the Boolean attribute icon 234.

Referring to **Figure 17**, the retriever 130 then proceeds to step 301. A Boolean dialog box 283 pops up to present true and false choices to the user. The Boolean dialog box 283, shown in **Figure 18**, includes a true option button 284 and a false option button 285. As is typical in a windows operating environment, these option buttons are mutually exclusive. The user may select either a true or a false search criteria 282 by clicking on the true option button 284 or the false option button 285, respectively.

The Boolean dialog box 283 includes an OK button 286. The user confirms the input in step 302 shown in **Figure 17** by actuating the OK button 286.

The Boolean dialog box 283 also includes a cancel button 287 and a clear button 288, which function in a manner known to those skilled in the art.

When the user confirms the desired Boolean search criteria 282 by actuating the OK button 286, the retriever 130 proceeds to step 303 in **Figure 17** and adds or replaces an appropriate

Boolean selector in the current query. As explained above, a Boolean selector will be added if no such selector existed in the previous query. A Boolean selector will be replaced if a Boolean selector appeared for this attribute 203 in the previous query. Flow then proceeds to step 210 in **Figure 17**. The Boolean dialog box 283 disappears, and the selected search criteria is displayed in the appropriate search criteria field 282 in the right hand portion 175 of the part specification window 170. The display of parts found 172 will also be updated.

Referring to **Figure 19**, a user may select an enumerated attribute 289 by clicking on an enumerated attribute icon 233. This is accomplished in step 305 depicted in **Figure 19**. The retriever 130 then displays an enumerated attribute dialog box 291, as shown in **Figure 20**. The enumerated attribute dialog box 291 presents a list of valid values the particular attribute 289 may have. The selected value is displayed as a highlighted entry 293. A user may select multiple values 292 for the search criteria 290. When multiple values 292 are selected, the retriever 130 treats them as an "or" logic condition for the search criteria 290. In other words, the search will retrieve parts that have any one of the enumerated values 292 which are selected.

The dialog box 291 includes a clear button 296. The user may deselect all of the selected values 293 and start over. This is convenient when multiple values 292 have been selected. The user may abort this operation by actuating a cancel button 295 provided by the dialog box 291. This will transfer flow from step 307 to step 309 shown in **Figure 19**.

When the user is satisfied with the selected enumerated values 293, the user can confirm the input by actuating an OK button 294 provided in the dialog box 291. Referring to **Figure 19**, flow proceeds from step 307 to step 308 when this occurs. The retriever 130 will add or replace the selected enumerated values 292 in the current search query, depending upon whether pre-existing values for this attribute have been selected in a previous query. The retriever 130 will then proceed to step 210 shown in **Figure 19** and update the display. The enumerated attribute dialog box 291 will disappear. The selected search criteria 293 appear in the search criteria display field 290.

retriever 130 will then proceed to step 309 shown in **Figure 19**, and return control to the user awaiting another command.

Referring to **Figure 20**, the enumerated attribute dialog box 291 also includes scroll buttons 297 which operate in a manner known to those skilled in the art.

Figure 21 depicts an updated display of the part specification window 170. In the illustrated example, the text attribute search criteria 242 is displayed in the search criteria display field associated with the part number attribute 241. The enumerated attribute search criteria 290 is displayed in the search criteria field associated with the enumerated attribute "head style" 289. The numeric search criteria 281 is displayed in the search criteria display field associated with the numeric attribute for length 236. The updated parts found display 172 reveals that the search has succeeded in locating a single part that has the desired attributes.

A flow chart of the procedure employed by the retriever 130 to determine the order in which the search results are to be displayed is depicted in **Figure 22**. The procedure starts with step 310 in which the user is allowed to select the order in which the display results are to be sorted according to the attributes of the parts located in the search. The display may be sorted according to attributes 176 which were not selected as search criteria, as well as attributes 241, 289, and 236 which were. Selection is accomplished by clicking on buttons 298 in the order column 194 to correspond to the desired attribute 241. The first order button 298 that is clicked will be the first attribute 241 with respect to which the search results will be sorted for display. A "1" appears on the display of the order button 298 which is clicked first. Similarly, the second order button 299 which is clicked will be the second criteria from which the search results are sorted, and a "2" appears on the display of the button 299.

When a user clicks on an order button 298 as shown in **Figure 21**, the retriever 130 proceeds to step 311 in the flow chart illustrated in **Figure 22**. The retriever 130 checks to determine whether the requested order button 298 is already set, i.e., already has a number on the order button 298. If it is not, flow proceeds to step 312 and the retriever 130 sets the order button

298 to the highest order currently set plus one. That is, if the user clicked order button 357 shown in **Figure 21**, the display order for that cost attribute 358 is not currently set. In this example, the highest order currently set is "6". Thus, in step 312 the order button 357 would be set to "7", because that is one more that the highest order currently set (i.e., 6+1=7). A "7" would then be displayed on order button 357. The flow then proceeds to step 315. Referring again to step 311 shown in **Figure 22**, if the display order is currently set for a particular order button 299, the flow proceeds to step 313 and the selected display order is unset. In other words, the display button 299 may be toggled off. Flow then proceeds to step 314, and each of the order buttons 361, 358, 359, and 360 currently set which have a number greater than the number of the order button 299 that was toggled off, will be decremented by one and reset. In the example illustrated in **Figure 21**, if the user clicks on the order button 299 (which currently displays a "2"), that order button 299 will be reset (it will appear blank like order button 357). In addition, the remaining order buttons 358, 359, 360 and 361 which have a currently set display order greater than order button 299 (i.e., greater than "2"), will be decremented by one. Order button 358 will be changed from "3" to "2", order button 359 will be changed from "4" to "3", and so forth. Order button 298 will not be changed, because its display order is "1" and is not greater than the display order of order button 299 which was reset. Flow then proceeds to step 315 and control is returned to the user.

The procedure for requesting display of search results is depicted in **Figure 23**. The procedure is initiated in step 316 when the user clicks on the display button 179. The procedure then moves to step 317 shown in **Figure 23**. In this step, the system does a query and obtains the query result. This step is shown in more detail in **Figure 25**. After the query result is obtained, the procedure then moves to step 318 and displays a search results window 299, an example of which is shown in **Figure 24**.

Referring again to **Figure 23**, the next step in the procedure is step 319. For each attribute specified in the display order 194 (as a result of the user clicking the associated display

order buttons 298, 299, etc.) a display column is created. The display columns are created left to right in the order specified by the order buttons 194. The procedure then moves to step 320, and for each part in the query result, the attribute values for the specified attributes are displayed in the respective display columns. Control is then returned to the user in step 321 and the retriever 130 waits for another command.

Referring to **Figure 25**, the procedure for doing a query is illustrated in more detail. The procedure starts at step 322. The system first determines in step 323 whether the query is a local query or a global query. If the query is a local one, the retriever 130 makes a list of parts in the selected class, as indicated in step 325. If the query is not a local one, the retriever 130 makes a list of all of the parts included in the selected subtree, as indicated in step 324. In either event, the system then proceeds to step 326 to determine whether unprocessed parts remain in the list. If the answer is yes, the flow proceeds to step 328 and the system gets the next part in the list. The procedure then goes to step 329 to determine whether all attribute selectors are matched by corresponding part parameters. If the answer is no, the procedure loops back to step 326. If the answer is yes, the procedure goes to step 330. The part is added to the query result in step 330, and the flow loops back to step 326. Referring back to step 326, if no unprocessed parts remain in the list, the procedure then proceeds to step 331 and returns the query result and part count.

Referring to **Figure 24**, the search results window 299 displays the selected attributes for the parts found in the search. The part number attribute 336 appears in the left most column because the order button 298 was selected first in this example. The finish attribute 337 appears second, because the finish button 299 was selected second. Similarly, the head-style attribute 338, the head recess attribute 339, the length attribute 340, and the description attribute 341 are displayed in the order indicated by the order buttons 194. In the illustrated example, the search had narrowed down the parts found 172 to a single part. If more than one part had been obtained in the search results, the remaining parts would be displayed in additional rows in the search results display window 299, and the

display would be sorted according to attributes in the order indicated by the order button 194.

A significant performance optimization of this invention concerns the management of a query result to optimize use of network resources, thereby allowing effective access to a knowledge base server 132 through a wide area network 2103, which typically has significantly lower transmission speeds and data throughput than a local area network 100. This is accomplished as shown in the flowchart in Figure ZZZ. In response to a user request to scroll the scrolled list 352 in step 2130 in either direction, buffers containing part information in the direction being scrolled is examined in test 2131 to determine if the scroll request results in a need for part information not currently in the buffer. If it does, then the buffer is refilled with sufficient part information from the query result to allow for scrolling of one additional display page of information without requesting additional information from the knowledge base server 132. After scrolling the display, parts information is displayed from the display buffer in step 2132 and control is returned to the user in step 2133. In this way, the network transmission cost that would have been incurred if the entire query result were transmitted to the server initially is avoided, significantly improving response time to the point where a wide area network 2103 provides a practical alternative to a local area network 100. This optimization also reduces overall network traffic and removes the need for any limits on the number of parts that may be displayed in a query result as are typically found in query systems.

Referring to **Figure 24**, the search results window 299 includes a part info button 342. The procedure initiated by actuation of the part info button 342 is shown in **Figure 26**. In step 332, the user clicks on the part info button 342. The system proceeds to step 333 to produce a display of a class path in outline format 350 from the root class 173 to the owning class 240 of the part, as shown in part information display window 351 in **Figure 27**. Referring to **Figure 26**, the procedure flow proceeds to step 334 and produces a display of a scrolled list 352 containing attribute names 353 and values 354.

Figure 27 depicts the part information window 351. The

attribute name 353 and values 354 for those attributes are displayed in a scrolled list 352. The part information window 351 shown in **Figure 27** may be closed by actuating the OK command button 356. Referring to **Figure 26**, the procedure then goes to step 335 and control is returned to the user.

The search results window 299 includes a user action command button 343. This user action button 343 is used to launch other user applications or software programs. This provides transparent access to other applications directly from the system. The user action command button 343 becomes active when a part in the search results window 299 is selected by clicking on the row number for that part, and a user action is associated with that part.

For example, it may be desirable to see an actual part drawing for a selected part. A CAD or viewer application may be selected from a pull-down list 344 of applications by clicking on button 345 and then clicking on the desired application included in the pull-down list 344. The desired user application is first selected, and then the user action command button 343 is actuated to cause the application to start and the designated file to open. User actions may be defined by the system administrator.

Figure 28 depicts a flow chart for the procedure used to launch a user action. In step 365, the user selects a user action from a list 344 on the search results window 299. The flow then proceeds to step 366, and the system looks up arguments to the user action by checking associated user action definitions.

In step 367, the system formats a command line with parameters filled in from part attributes 336, 337, and 340 specified in user action definitions. In step 368, a local process is executed using the formatted command line and block until the user action is completed and the process exits. Finally, in step 369, control is returned to the user.

In the example illustrated in **Figure 24**, the Microsoft Windows Write program 344 has been selected. When the user actuates the user action button 343, the Write program 344 starts. In this example, the part number 336 is passed to the Write program 344. **Figure 29** shows the user action display

screen 355 when the write program 344 starts, where the part number information 336 was passed to the Write program.

The search results window 299 includes a sort command button 348. In the illustrated example, this button 348 is dimmed because only one part is displayed. When a plurality of parts are displayed in the search results window 299, the sort command button 348 may be actuated to re-sort the displayed information differently.

The search results window includes a print command button 347. Actuating this button 347 causes a hard copy print out of the parts to be generated. This is convenient, for example, when the user wants to do some additional research on the parts.

The search results window 299 includes an apply command button 346. This button 346 is used to copy the attribute values for the selected part to the search criteria fields 177 in the part specification window 170. After the values are copied, the search results window 299 closes. The results of actuating the apply command button 346 are shown in **Figure 31**. This may be conveniently used, for example, to conduct another query in which one of the parameters is relaxed.

The procedure executed when the apply command button 346 is actuated is shown in **Figure 30**. The procedure starts at step 370 when the user actuates the apply command button 346. Step 371 is then executed, and a new query is created with the selected parts owning class as the query class. In step 372, an appropriate attribute selector is added to the query for each defined attribute for the part.

The part specification window 170 is displayed in step 373 with the class outline 248 open to the class 240 of the current query. Then in step 374 the attribute selectors 242, 281, etc., are displayed for the current query. The system then updates the part count 172 and the display 170. Control is returned to the user in step 375.

Alternatively, the search results window 299 may be closed by actuating close button 349.

A user who has the necessary access rights may edit parts information in the knowledge base by actuating the edit command button 180. The procedure that is executed is shown in **Figure 32**. Step 376 is executed when the edit button 180 is selected.

In step 377, a query is performed based upon the current search criteria 177 and the retriever 130 gets the query results. The query results are then displayed in a spreadsheet format in step 378.

In step 379, the system then handles part move, delete, and attribute edit requests.

The sort procedure executed when the user actuates the sort button 348 in the search results window 299 is depicted in the flow chart of **Figure 33**. Step 380 is performed when the user clicks on the sort button 348. The system then displays a sort dialog box 386, as described in step 381.

An example of a sort dialog box 386 is shown in **Figure 34**. Attributes 387 through 392 are displayed in an attribute column 393 in accordance with step 381 of **Figure 33**. The dialog box 386 also includes a sort key column 394 and a sort order column 395. The sort order column contains pull-down menus actuated by appropriate buttons 396 (only one of which is shown) which allow the user to select ascending order or descending order for each attribute 387 - 392. This is described in step 381 shown in **Figure 33**.

The sort procedure allows a user to reorganize the list of parts in an alphanumeric or numeric sequence. The user can sort the list of parts in ascending or descending order based upon one or more of the attribute values 387-392. The sort key 394 identifies which attribute 387-392 the user wants to sort by first, which attribute the user wants to sort by second, which attribute the user wants to sort by third, and so on.

For example, if the user has a list of parts and the user wants to obtain a sorted listing of these parts according to one of the attributes 392, the user selects that attribute 392 as the first sort key. The user may select a second attribute 389 to sort on when the first attribute 392 has duplicate values. In the example shown in **Table 3**, the length attribute 392 was the first key and the order was ascending. The major material attribute 389 was not selected as a sort key.

Table 3

Length	Major Material
.5 (Inch)	Phosphate
.5 (Inch)	Zinc

.5 (Inch)	Phosphate
.75 (Inch)	Phosphate
.75 (Inch)	Zinc
.75 (Inch)	Phosphate

Notice that in the example shown in **Table 3**, there are duplicate values in the length column and because of this, the values in the materials column are random. If the user wants these values to be sorted in conjunction with the first key, the user would select the major material attribute 389 as the second sort key. In this example, for both attributes 392 and 389 the user would use the default sort order, which is ascending. The example shown in **Table 4** is the result when using this type of sort.

Table 4

Length	Major Material
.5 (Inch)	Phosphate
.5 (Inch)	Phosphate
.5 (Inch)	Zinc
.75 (Inch)	Phosphate
.75 (Inch)	Phosphate
.75 (Inch)	Zinc

Selecting ascending as the sort order causes the order of the attribute values to be sorted in a manner which depends on the type of attributes where enumerated attributes are sorted, the enumerated attributes with a value of "undefined" are listed first, then the remainder of the values are listed in the same order as they appear in the schema. When text attributes are sorted, the text attributes with a value of "undefined" are listed first, then the values that are numeric, and then the text is sorted in ASCII sequence. When numeric attributes are sorted, the numeric attributes with a value of "undefined" are listed first, then the values that are numeric based upon the unit of measurement. When boolean attributes are sorted, the boolean attributes with a value of "undefined" are listed first, then the attributes with a true value, and then the attributes with a false value.

Selecting descending as the sort order causes the order of

the attribute values to be reversed.

To establish the sort order the user wishes to use, the user should choose the sort command button 348, then from the sort dialog box 386, select the attribute 392 the user wants to sort first, then choose the set command button 398. Choosing the set command button 398 sets the key 394 and the sort order 395 for the attribute 392. Double-clicking in the key field 394 of the attribute 392 sets both the sort key 394 and the sort order 395.

To clear the key 394 and sort order 395, the user may select the attribute 392 that the user wants cleared, then choose the set command button 398. The key 394 and sort order 395 for the selected attribute 392 will then clear. As shown in step 382 in **Figure 33**, the user can cancel the input by actuating the cancel command button 397. If the user does so, flow will jump to step 385 and control will be returned to the user.

After selecting all the attributes 387-392 with respect to which the user wants to sort, the user may actuate the OK command button 399. This results in flow going to step 383 in **Figure 33**. In step 383, the query result is sorted according to the requested compound sort key. In accordance with step 384, the sort dialog box 386 will close and the search results window 299 will reappear with the parts information sorted according to the user's selections. Then in step 385, control will be returned to the user.

The user may be required to edit the parts in the user's knowledge base whenever the user has additional data to further classify a part, there is a duplicate part, or if the user needs to move parts from one class to another class.

The edit command button only appears in the parts specification window when the user has access rights for this feature. The system administrator is responsible for setting access rights to this feature. There are two features that will assist the user in locating parts that need editing. One is searching for parts that have undefined attributes. The other is using the local query command to find parts that have not been fully classified.

From a parts editor window 1019, the user can edit the attribute values, move parts from one class to another, and delete parts.

In order to edit parts, the user follows the same procedures that the user uses for specifying a part. In addition to specifying parts with specific attribute values 1056, the user can also locate the parts that do not have values (undefined) for a specific attribute 1060. To locate parts with undefined attribute values, the user selects the undefined check box 165 for the specific attribute 166. See **Figure 6**. When the undefined check box 165 contains a check mark, it is selected, indicating that the user is searching for parts that do not have that attribute 166 defined.

The user would use undefined if the user is editing the data in the user's knowledge base and the user wants to locate attributes 166 that are currently undefined 1060 so the user can research those parts and update the knowledge base to include the appropriate values.

If the user selects a specific attribute value and also selects "undefined", the user sets up an "or" condition as part of the user's search criteria. In this example, the system locates parts with the specific value and the parts that do not have a value for that attribute. The undefined check box 165 is positioned to the right of the search criteria field 177. The user may need to use the horizontal scroll bar to move the check box 165 into view.

Once the user has specified the part by selecting the class 174and subclasses 196, 197, 198, and 199, entered the attribute search criteria 177, and set the display order 194, the user can choose the edit command button 180.

Figure 35 depicts a flow chart showing the procedure followed when a user edits parts. Referring, for example, to **Figure 21**, a user who has access rights to edit parts may actuate the edit button 180 and bring up the parts editor window 1019 shown in **Figure 36**. The first step 1012 shown in **Figure 35** involves the user selection of attributes and parts to edit from the parts editor window 1019. A user may enter new or revised values 1061for attributes 1051, and the system will accept parameter input in step 1013. If the attribute is an enumerated attribute 1051, a pull down list 1062 will be presented to the user with available choices, as shown in **Figure 37**. In step 1014 of **Figure 35**, a determination is made as to whether there are

more parts to edit. If there are no more parts to edit, flow proceeds to step 1017. The system updates the part display 1020 and the parts editor window 1019 with edited values 1061. The system then proceeds to step 1018 and returns control to the user.

In step 1014, if more parts remain to be edited, flow proceeds to step 1015, and the system gets the next selected part. In step 1016, the system sets the next selected parts parameter to the user input value 1061. Control then loops back to step 1014.

Figure 38 depicts a procedure for deleting parts. In step 1021, the user selects parts to delete from the edit parts window 1019. The user then clicks a delete parts command button 1026. In step 1022, a determination is made as to whether any more parts remain to be deleted. If the answer is yes, flow proceeds to step 1023 in which the system gets the next selected part and deletes it from the query result and the knowledge base. Flow then loops back to step 1022. When there are no more parts to delete, flow proceeds to step 1024, and the system redisplays the updated query result in the part editor window 1019. Flow then proceeds to step 1025, and control is returned to the user.

Figure 39 depicts a flow chart for a procedure for moving parts. The procedure may be initiated by the user selecting parts to move from the parts editor window 1019 as shown in step 1032. Alternatively, the user may initiate the procedure as in step 1033 by navigating the class hierarchy on the parts editor window 1019 and selecting a destination class. The user may actuate a move parts command button 1027, which is illustrated for example in **Figure 40**.

Referring to **Figure 39**, the procedure proceeds to step 1034 and a determination is made as to whether there are more parts to move. If there are no more parts to move, flow transfers to step 1042 and the system redisplays the query result in the parts editor window 1019. The flow then proceeds to step 1043, and control is returned to the user.

Returning to step 1034 in **Figure 39**, if a determination is made that there are more parts to move, flow proceeds to step 1035 and the system gets the next selected part. In step 1036 a determination is made as to whether the user has requested an

unconditional move. If the answer is yes, flow jumps to step 1040. The system then sets the part class to the destination class selected by the user. Any parameters or missing attributes are set to undefined. Flow proceeds to step 1041, and the system deletes the moved part from the query results. Flow proceeds to step 1042 where the system redisplays the query result in the parts editor window 1019.

In step 1036, if the user has not requested an unconditional move, flow proceeds to step 1037 where a determination is made as to whether attributes for any part parameters are missing from the destination class. If the answer is no, flow proceeds to step 1040 and continues as described above.

If a determination is made in step 1037 that there are attributes for part parameters which are missing from the destination class, flow transfers to step 1038. The system gets a list of parameters that will be deleted by the move and presents them to the user by displaying them on the display 116. Flow then proceeds to step 1039. If the user then overrides the warning that parameters will be deleted, or requests that the parts be moved unconditionally, flow transfers to step 1040 and proceeds as described above. If the user does not wish to override the parameter deletion warning or does not request that the parts be moved unconditionally, flow loops back to step 1034.

The process of editing parts may be further understood in connection with a description of the parts editor window 1019 (shown in **Figure 40**). Once the user has specified a part by selecting a class 174 and subclasses 196, 197, 198 and 199, entered the attribute search criteria 177, and set the display order 194, the user can edit the parts by choosing the edit command button 180. Choosing this command 180 causes the parts editor window 1019 to appear. The top area 1052 of the parts editor window 1019 contains the class tree 1044, showing highlighting the navigation path and class definition of the parts the user is editing. The bottom area 1053 of the window 1019 contains the parts 1020 the user has selected to edit. The parts appear in a table 1020 that is similar to tables that are used in spreadsheet applications. The part attributes 1049, 1050, 1051, etc., and attribute values 1055, 1056, 1057, etc., appear in the display order, from left to right, that the user

previously established in the part specification window 170. To use a value, the user clicks an enter box 1063. To cancel a new value, the user clicks a cancel box 1064.

The top area 1052 of the parts editor window 1019 contains the class definition 1044, which comprises the class tree showing the navigation path and class definition of the parts selected for editing. The window 1019 has a horizontal split bar 1047 that splits the window into two sections 1052 and 1053. The user can move the split bar 1047 up or down so the user can see more of one section 1052 or the other 1053. The parts editor window 1019 includes an area referred to as the editing area 1046. After selecting an attribute value 1051, a text box or list box 1054 appears in this editing area 1046 so the user can make changes (see **Figure 36**). Each part appears as a row 1048 in the table 1020,and each row 1048 of the table 1020 is numbered. The user may use the row number to select a part that the user needs information on or that the user wants to move or delete. The attributes 1049, 1050, 1051, etc., are the column headings, and the attribute values are the rows.

Referring to **Figure 40**, a sort command button 1029 may be actuated to rearrange the parts according to a sort order that the user may enter. A part info command button 1028 may be actuated to display all of the part information (the class definition and all attributes) for a selected part. A print command button 1030 may be actuated to print the list of parts. A delete command button 1026 becomes active after the user selects a part, and can be used to remove obsolete parts from the user's knowledge base. A close command button 1031 may be actuated to close the parts editor window 1019 and return focus to the part specification window 1070.

In order to make a part, a user follows the same procedures the user uses to specify a part. If the user specifies a part that results in zero parts found 172 and an acceptable substitute does not exist, the user can add a new part to the knowledge base.

After determining that the user is going to enter a new part in the knowledge base, the user must fully specify the part. In a preferred embodiment, a complete part specification is defined as selecting the class up to the leaf class 201 and entering

values for all the required attributes 203. In a preferred embodiment, if the user does not select a leaf class 201 or enter the required attributes 203, the user cannot add the part. When making parts, a preferred procedure is for the user to enter as many attribute values 203 as the user can in order to give the part as complete a specification as possible.

Some attributes are required before a part can be added. Required attributes have a required icon 263 immediately to the left of the attribute icon. Before choosing the make command 181, the user must enter an attribute value for each required attribute. In addition, a user cannot enter any attribute values for protected attributes. Protected attributes have a protected icon 191 immediately to the left of the attribute icon. Once the user has selected the leaf class 201 and entered all required attributes, the user can choose the make command button 181. Choosing the make command 181 causes the part to be added to the user's knowledge base and the parts found 172 to be updated to show a part count of 1.

Although the above description has been with reference to a Windows client 112, the system is not so limited.

B. Knowledge Base Client

The knowledge base client 131 is a set of C++ libraries that provide knowledge base services to a client application 130, 133, and 144 through the API 143. The services may be either local or result in remote procedure calls to the knowledge base server 132. For client applications which run under Windows, the knowledge base client consists of one or more Windows Dynamic Link Libraries (DLL) which use the WinSock DLL to provide network access to the knowledge base server 132 and the registry server 141.

C. Knowledge Base Server

The knowledge base server 132 is a UNIX server process that manages knowledge base 103 access, retrieval and updates. A knowledge base server 132 may manage one or more knowledge bases 103 and 105.

1. Dynamic Class Manager

The dynamic class manager 134 is a software subsystem in the knowledge base server 132 that manages schema and data. The dynamic class manager 134 provides the ability to store class,

attribute, unit and instance information that can be modified dynamically. The dynamic class manager 134 consists of C++ libraries and classes and provides operations for "legacizing" and for accessing, creating, deleting, and modifying classes, attributes, instances, parameters, unit families, units and meta-attributes at run-time.

The capabilities of the dynamic class manager 134 are accessed by a user programmer through a set of functions provided by the API 143.

The dynamic class manager 134 knowledge base, hereafter sometimes referred to as "the knowledge base," is a collection of classes, attributes, units, instances with parameter values, and relationships among these objects. In the dynamic class manager 134, a class defines a separate type of object. Classes have defined attributes. The attributes have some type, and serve to define the characteristics of an object. A class can be derived from another class. In this case, the class inherits attributes from its ancestors. A knowledge base contains instances of classes. The attribute values defined by an instance are parameters.

Another way to describe the concept of classes, attributes, instances, and parameters is to use a dog as an example. The word "dog" is the analog of a class. Dog describes a group of similar things that have a set of characteristics, or attributes. The attributes of a dog are things like color, breed, and name. The class and attributes do not describe any particular dog, but provide the facility to describe one. An instance of a dog has parameters that give values to the attributes: for example, a dog whose color is beige, of the breed golden retriever, and whose name is Sandy.

Classes can have relationships. The class "dog" is part of the larger class, "mammal". The class "mammal" is less specific than "dog". It conveys less information about the object "dog", but everything about "mammal" also applies to "dog". "Dog" is clearly a subset of "mammal", and this relationship is a subclass. "Dog" is a subclass of the class "mammal". The subclass "dog" could be further subclassed into classes like big "dogs", little "dogs", etc. The concept subclass implies a parent relationship between the two classes. "Mammal" is a

parent and "dog" is a subclass. The terminology "'dog' is derived from 'mammal'" is also used to describe the relationship.

The subclass "dog" inherits attributes from its parent class. The attribute color could be part of the "mammal" class, since all "mammals" have a color. The "dog" class inherits the attribute color from its parent.

The root class is special, it has no parent. It is the class from which all classes begin their derivation. In illustrations set forth herein, graphs have been drawn to illustrate a class hierarchy, and the root class is placed at the top of those drawings. Subclasses branch out from the root class into ever widening paths that make the graph look like an upside down tree. The entire group of classes is a tree, and the special class that has no parent, though it is at the top, is the root.

One of the available attribute types supported by the dynamic class manager 134 is a numeric type. Numeric attributes are used to describe measurable quantities in the real world. Such measurements do not consist of just a numeric value; they also have some associated units. The dynamic class manager 134, in conjunction with the units manager 138, maintains information about different types of units that can be used with numeric attributes. The dynamic class manager 134 (using the units manager 138) can also perform conversions among units where such conversion makes sense. The units that the system understands are grouped into various unit families. These unit families and the units they define, can be changed at run time. The dynamic class manager 134 also comprises a dynamic units manager 138.

The word "schema" refers to the layout of classes, attributes, units, and unit families. A knowledge base with no instances is a schema. This may be better understood in connection with the following more detailed description of the various objects managed by the dynamic class manager 134.

A class is the most fundamental object in the schema in accordance with the present invention. A class is a collection of related objects. In the present example, a class may have eight or nine components. A class is a schema object. As explained above, the schema is the collection of classes, attributes, units, and unit families and their relationships.

Every class has exactly one parent from which it is derived, except for the root class 173. The root class 173 is the one class that has no parent. The root class 173 has another special characteristic in that it can never be deleted. The consequence of a class being derived from its parent means that the class has all of the properties of its parent. These properties are referred to as attributes. Attributes are inherited from the parent class.

A class may have zero or more subclasses. A class is a parent of each of its subclasses. A subclass is a class that has a parent, so the root class 173 is not a subclass. The subclasses of a parent class have some defined order. The order is persistent, meaning that the dynamic class manager 134 preserves the order even across closes and reopens of the knowledge base.

A class has a set of descendants that is comprised of all of its subclasses, all of the subclasses' subclasses, and so on. A class that has zero subclasses or an empty set of descendants is called a leaf class 201. A subtree is the set composed of a class and all of its descendants. The subtree is said to be rooted at the class. A subclass also has a set of ancestors, which is the set composed of the parent, its parent's parent, and so on including the root class 173. Classes that have the same parent are sometimes referred to as siblings.

Following a subclass to its parent is sometimes referred to as going up the tree. Moving from a parent to one of its subclasses is sometimes referred to as going down the tree. Therefore, the root class 173 of the schema is the furthest up at the top of the tree, and the objects furthest down at the bottom of the tree are typically leaf classes 201.

A class has a name which is the text identifying a class, subclass, or leaf class, and is an ASCII character string. The present invention uses class handles for references to a class, which are further described in connection with the operation of the handle manager 137. In the example shown in **Figure 5**, there are three subclasses.

Figure 41 shows the internal object representation for a class 800. In the present schema, a class has a parent handle 801. Every class object 800 includes stored information

representing the handle of its parent class, except in the special case of the root class 173, which has no parent. A null is stored in this location in that case. A handle is a reference to an object. The parent handle information 801 is used by the handle manager 137 to identify the stored class object which is the parent class for the class 800.

The class object 800 includes a subclass list 802. The subclass list 802 is an array of handles which may be used by the handle manager 137 to identify those class objects which are subclasses of the class 800. In the internal representation provided in the present invention, lists can grow without bounds and are dynamic. The storage space available is not fixed.

This provides flexibility and power to the database structure, because the class object 800 may have an extremely large number of subclasses in a large database without substantial degradation in performance.

The class object 800 includes an attribute list 803. The attribute list 803 is a list of handles. The handle manager 137 may use the information stored in the attribute list 103 to identify the attributes possessed by class object 800.

The class object 800 also includes a local instance list 804, which is a handle list. Field 805 shown in **Figure 41** is a pointer to storage location of the class name, i.e., the text identifying the class.

Field 806 is used to store the handle for the class 800. The field 807 stores an indication of the class code, i.e., whether it is primary, secondary, or a collection.

The class object 800 also includes a subtree instance count 808. The subtree instance count 808 is a numeric indication of the total number of items or instances present in all of the descendants of the class 800 i.e., the total number of instances in class 800, all of the class 800's subclasses, all of the subclasses' subclasses, and so on. Referring, for example, to **Figure 10**, the instance count 808 is used to generate the parts found 172 field which is displayed on the part management window 170. Thus, when a user is navigating through the tree structure of a knowledge base, as a user selects and opens subclasses, the user can be immediately informed of the number of parts found 172 at any location on the tree by retrieving the subtree instance

count 808 for the current class and passing that information to the retriever 130. The subtree instance count 808 is kept up to date whenever the knowledge base is modified, so it is not necessary while a user is navigating through the tree structure of the database to perform a real time computation of parts found 172.

Referring again to **Figure 41**, the class object 800 also preferably includes a metaparameter list 809. The metaparameter list 809 is a string list, and may be used as a pointer to strings containing linking information ,for example, the name of a file that contains a graphical display of the type of parts represented by the class 800, thesaurus information used for legacizing data, or other legacizing information.

Figure 42 depicts an example of a generic list 810. The class manager 134 uses lists of handles, lists of floating point values, lists of pointers to character strings, etc. whenever a variable amount of data can be associated with an object. Examples of lists would be items 802, 803, 804 and 809. The list 810 depicts a
list of simple integers.

A list object 810 includes a pointer 812 which points to the beginning 815 of the list data 811. A list object 810 also includes a field 813 indicating the currently allocated size for the list data 811. The list object 810 also includes a field 814 containing information indicating the amount of list data 811 currently in use.

The list data 811 contains the actual list of values. The first item 815 in the list in this example contains the value "5". Similarly, in this example list items 816, 817, 819, 820 and 821 contain additional values. List items 822, 823, 824, 825 and 826 in this example are not currently in use and are set to zero. In this illustrated example, the currently allocated size 813 of the list is twelve. The amount in use 814 of the list is seven, meaning that the first seven items in the list are valid.

Figure 43 illustrates the data structure for attribute data 827.

An attribute object 827 contains at least six fields in the illustrated embodiment. A first field 828 contains a pointer to

an external name comprising an ASCII character string that is the name for the attribute. The attribute object 827 also contains a field 829 containing the handle for this attribute object 827. The attribute object 827 also contains a field 830 which contains the handle of the class that defines this attribute 827. The fourth field 831 is a Boolean indication of whether this attribute is a required attribute for the defining class. A fifth field 832 contains a Boolean field indicating whether this attribute is protected. For example, in **Figure 6** the "part number" attribute 176 is protected. This is indicated by the protected icon 191. In the data structure of the attribute object 827 shown in **Figure 43**, this information is stored in field 832. The attribute object 827 also contains a field 833 which is a metaparameter list.

Enumerated attributes include fields 828 - 833, indicated collectively as attribute data 834, plus a field 835 which is a list of enumerator handles.

In the case of a Boolean attribute, only fields 828 - 833 are used, which are again indicated collectively in **Figure 43** as attribute data 834.

Numeric attributes include fields 828 - 833, indicated collectively as attribute data 834, plus a field 838 which contains the handle of the unit family for this numeric attribute.

In the case of a string attribute, and in the case of a string array attribute, only the attribute data 834 comprising fields 828 - 833 is included.

One example of the use of these data structures by the dynamic class manager 134 is the procedure of a user selecting a class by clicking on the closed folder icon 189 associated with the class (see **Figure 7**). When a class is opened, the dynamic class manager 134 will check the class object 800 and retrieve the attribute list 803. The handles stored in the attribute list 803 will be passed to the handle manager 137. The handle manager 137 will return the virtual memory address for each attribute 827 of the class. The dynamic class manager 134 may then use the pointer 828 to the external name of an attribute object 827 to retrieve the character string text for the external name for the attribute. That ASCII text information can then be passed

through the API 143 so that it may eventually be provided to the retriever 130 for display to a user on the display 116.

Figure 44 illustrates the data structure for an enumerator object 841. An enumerator object 841 may comprise three fields. A first field 842 contains a pointer to the external name for the enumerator object 841. A second field 843 contains the handle for the enumerator object 841. A third field 844 may contain a metaparameter list. Handles are used to link from other objects to the enumerator object 841. An advantage of this structure is the ability to easily modify a knowledge base if it becomes desirable to change the external name of an object. Such a change need only be performed once to the ASCII character string that is used to represent the external name. All other objects merely contain a handle which can be used by the handle manager 137 to provide the dynamic class manager 134 with the actual external name.

Figure 45 depicts the data structure for a unit family object 845. In the example illustrated in **Figure 45**, the unit family object 845 has four fields. The first field is a pointer to the external name 846 for the unit family object. The second field 847 contains the handle for this unit family object 845. The third field 848 is a list of unit family handles of unit families which are included in the unit family 845. The field 849 contains a list of handles for local units.

A unit is a system of measurement for a numeric parameter. A unit family is a collection of units that may be used for a numeric attribute. A unit family handle is a reference to a unit family. A unit family name is the ASCII text that identifies a unit family. A unit handle is a reference to a unit. A unit name is the ASCII text that identifies a unit. Local units are units that are defined in this unit family 845.

Figure 46 depicts the data structure for units. A unit object 850 may comprise five data fields 851 - 855. The first field 851 is a pointer to the external name for the unit. The handle for the unit object 850 is stored in the second field 852. The third field 853 contains the handle for the defining unit family. The fourth field 854 is a metaparameter list. The last field 855 contains an indication of the type of unit (e.g., real, integer or enumerated table). These five fields 851 - 855

comprise the base unit data 856.

If the unit object 850 is a base unit, then no additional data is required. This is represented by item 862 in **Figure 46**. If the unit object 850 is an enumerated derived unit 867, it will contain the base unit data 856, which includes fields 851 - 855. An enumerated derived unit 867 will also include a field 858 which provides the handle for the base unit. Another field 856 provides information on how many rows are present in the enumerated list. The field 860 provides the list of enumerators which typically comprises ASCII character strings. The field 861 provides a list of corresponding values for the list of enumerators in field 860.

If the unit object 850 is a real derived unit 866, it will include the base unit data 856 which comprises fields 851 - 855. In addition, it will include a field 863 in which is stored the handle for the base unit. A second additional field 864 will contain a multiplication factor used to derive the real derived unit 866. A third additional field 865 will provide an offset, if any, which must be added or subtracted to the base unit in order to derive the real derived unit 866. For example, if the base unit 850 is degrees centigrade, and the real derived unit 866 is degrees Fahrenheit, the multiplication factor 864 would be 9/5 and the offset 865 would be 32 degrees.

Figure 47 depicts the data structures for a unit families.

The dynamic class manager 134 maintains a single global unit family handle list 836. One element 837 in that list is the handle for unit family 845. For simplification, an arrow has been drawn directly from the handle 837 to the unit family 845. In actual practice, the handle 837 from the list 836 is provided to the handle manager 137, and the handle manager 137 provides the address in virtual memory for the unit family 845. It should be understood therefore that the handle manager 137 is involved in linking handles to the objects associated with the handles. With the understanding that such linkage to the handle manager 137 occurs in every instance where a handle is used to refer to an object, further reference to the handle manager 137 in this description may be omitted for purposes of simplification. In addition, data fields and or data members that are unnecessary for purposes of this description have been omitted from **Figure**

47. In the example illustrated in **Figure 47**, the list of included unit family handles 848 is empty. The actual list of local unit handles 839 is pointed to by list object 849 in the unit family object 845. By going to the list of local unit handles 839, the dynamic class manager 134 can lookup the desired unit object. For example, item 857 in the list 839 is a handle which refers to real derived unit 866. In this example, the unit family 845 is "resistance", and the real derived unit 866 is "kohms". The real derived unit 866 has the handle of the base unit stored in field 863. The handle stored in field 863 is used to lookup the base unit 850, whose name 852 in this instance is "ohms". The real derived unit object 866 contains a multiplication factor 864, which in this example is 1,000. The offset 865 is zero. Thus, the units manager 138 will use the multiplication factor 864 to convert the derived unit "kohms" 850 to the base unit "ohms" 866 by multiplying by 1,000.

The real derived unit object 866 contains a handle 853 for the defining unit family 845. The unit object 850 also contains a field 853 containing the handle for the unit family 845.

Figure 48 shows the data structure for an enumerated derived unit. A global unit family handle list 836 may contain an item 837 in the list which is the handle for unit family 845 as described with reference to **Figure 47**. However, in this example, the global unit family handle list 836 also contains an item in the list 862 which is the handle for a second unit family 845'. The second until family 845' has a name 847'. The list of included unit handles 848' in this example has the handle for unit family 845. The unit family object 845', includes a data field 849' which points to a list of local unit handles 839'. The list 839' includes an item 868 in the list which is the handle for an enumerated derived unit object 867. In this example, the name of the enumerated derived unit object 867 is "table of ohms" 852. Field 859 contains information on the number of rows which are included in this enumerated derived unit object 867. Field 860 points to the list of enumerators 869 which lists the values that may be selected by the user from an enumerated list, in this example, "10k", "11k", "12k", etc. The list 869 contains strings of ASCII characters. In the enumerated derived unit object 867, the field 861 contains a pointer to the

list of real numeric data values 870. There is a one to one correspondence between the items in the list 869 and the numeric values in the list 870. In the illustrated example, the list 870 contains the actual numeric values 10000; 11000; 12000; etc. Because these values represent ohms in this example, they correspond to "10k", "11k", "12k", etc., ohms. Of course, the enumerated derived unit object 867 contains a field 858 which has the handle for the base unit object 850, which in this case has the name 852 of "ohms".

Figure 48 depicts the data structure that is used by the dynamic class manager 134 to provide information to the retriever 130 necessary to display a table of standard values window 273 as shown in **Figure 16.** In **Figure 16**, the plurality of standard values 275 comprise a display of the ASCII characters contained in list 869 shown in **Figure 48**. When one of the values 275 is selected by the user, the units manager 138 provides the dynamic class manager 134 with the corresponding numeric value taken from list 870 shown in **Figure 48**.

Figure 49 depicts the data structure for an instance 871 and associated parameters 872. An instance object 871 may contain four fields 873 - 876. The first field 873 is the handle for the owner class of this instance. The second field 874 may give the ordinal location of this instance's handle in the instance list 804 of its owning class. The third field 875 is a list of parameters, which points to the values contained in 877. The fourth field 876 is the handle for the instance object 871. The list of parameters 877 contains a plurality of pointers to parameters for the various attributes associated with this instance object 871. In the example illustrated in **Figure 49**, the list 877 contains three entries 878, 879 and 880. Additional elements of the list 877 have been omitted for clarity. The pointer 878 in list 877 points to information concerning the associated parameter 872. The data structure for the parameter 872 is illustrated in more detail in **Figure 50**.

Figure 50 shows the data structure for five different types of parameters: enumerated, Boolean, numeric, string and string array. Each of the parameter objects 872 has an attribute handle 881. An enumerated object 888 has an attribute handle 881 and an enumerator handle 882. A Boolean object 889 has an attribute

handle 881 and a Boolean value 883. A numeric parameter object 890 has an attribute handle 881, a unit handle 884 and a value 885. For example, if the numeric parameter is 10 ohms, the unit handle 884 would be the handle for the ohms unit, and the value 885 would be 10. A string parameter 891 contains a field for the attribute handle 881 and a pointer 886 to an ASCII character string. A string array parameter 892 contains an attribute handle 881 and a field 887 that points to a list of pointers to string arrays.

Figure 51 is an example of a schema with instances. The example has a class named "electronics", which has a subclass 800' named "capacitors". The capacitors subclass 800' has an attribute 827 called "case type". There are two possible types of cases in this example, which are referred to as "case A" and "case B". The subclass capacitors 800' has a subclass 800' named "electrolytic". The electrolytic subclass 800' has an attribute 827' called "voltage rating", and one instance 871' is provided that has parameters 890 and 888 of 5 volts and a type B case, respectively. Most objects and lists are shown incomplete in order to simplify the illustration, it being understood that like reference numerals refer to the same objects described in connection with **Figures 41 - 50**.

In **Figure 51**, the class object 800 has a name 806, which in this case is "electronics". The class object 800 has a field 802 which points to a list of subclasses 893. The list 893 has a first entry 894 which is the handle for the subclass 800'. In this case, the name 806' of the subclass 800' is capacitors. Of course, all references to schema objects actually use handles (not shown in **Figure 51**) and actually go through the handle manager 137 and handle table. This is not shown in **Figure 51** in order to simplify the diagram.

The subclass 800' capacitor has a field 802' which points to a list of subclasses 893'. The list 893' has an entry 894' which is the handle for subclass 800". The name 806" for subclass 800" is electrolytic. The subclass 800" has a null entry in the field 802" which would normally contain a pointer to a list of subclasses, if any. In this example, the subclass 800" does not have any subclasses.

Returning to the capacitors subclass 800', field 803

contains a pointer to a list of attributes 897. The list 897 contains the handle for the enumerated attribute 827 called "case type". Field 830 of the enumerated attribute object 827 contains the handle of the defining class 800' called capacitors. The enumerated attribute object 827 contains a pointer 835 which points to a list 839 of handles for enumerators. In this example, the list 839 contains a handle 898 for the enumerator 841. The enumerator 841 contains a pointer 842 to the external name for this enumerator, which may be an ASCII string for "case A". Similarly, item 899 in the list 839 points to enumerator 841' associated with case B.

Returning now to subclass 800" named electrolytic, the pointer 803" points to a list 897' of attributes, and one of the fields in the list 897' contains the handle for numeric attribute 827' which is "voltage rating". The numeric attribute 827' contains a field 830' which contains the handle of the defining class which in this example is the class 800" named electrolytic. The numeric attribute object 827' also contains a field 838' which contains the handle of the voltage unit family (not shown).

Returning to the electrolytic class 800", a field 804" contains a pointer to a list 895 of handles of instances. Item 896 in the list 895 contains the handle associated with instance 871. Instance 871 contains a field 873 which contains the handle of the owning class, which in this case is the electrolytic class 800". The instance data object 871 also contains a field 875 which points to a list of parameters 877. The list 877 contains a pointer 878 which points to the numeric parameter 890. The numeric parameter 890 contains a field 881 which contains the handle of the attribute 827' (voltage rating). The numeric parameter object 890 also contains a field 884 which has the handle of the units, which in this case is "volts". For simplicity, the unit object is not shown. The numeric parameter object 890 contains a field 885 which contains the value 5.0. In this instance, the electrolytic capacitor is rated at 5.0 volts.

The parameter list 877 contains a pointer 879 which points to the enumerated parameter 888. The enumerated parameter object 888 contains a field 881' which contains the handle of the attribute, which in this instance is case type. The enumerated

parameter object 888 also contains a field 882 which is the handle for the enumerator 841'. In this example, the electrolytic capacitor rated at 5.0 volts has a type case B.

The data structure described herein has significant advantages. Referring to **Figure 51**, it is easy to change a name or description in this data structure. Consider an example where the database may contain 1,000 instances of capacitors with a type B case. If the type B case is discontinued, or the name changed to "re-enforced", the only change that would need to be made would be to replace a single ASCII string representing the name for that case type. All 1,000 instances in the database simply contain a handle that the handle manager 137 associates with that ASCII text string. No other changes need to be made in the database.

Another advantage of the data structure in accordance with the present invention is that if a primary value is undefined, nothing is stored. Thus there is no wasted space.

Another advantage of the database structure is that algorithms do not have to be changed based upon location in the tree structure. All algorithms work the same regardless of location in the tree structure. The only special case is the root class. For example, the algorithm for adding an instance to the database is the same no matter where in the tree structure you are located. This makes dynamic changes to the schema very easy. A class or an entire branch of the tree structure can be moved from one location to another simply by changing lists of handles. It is not necessary to run a convert program. Everything is self contained. A class object 800 contains the handle of its parent 801 and thus knows who it's parent is. The class object 800 also contains a pointer 802 to a list of its subclasses, so it knows who its children are.

In the present database structure, it is possible to delete instances quickly. An instance can be deleted by taking the last item in the list of instances 804 and moving it to the position of the instance being deleted. In other words, the handle of the last instance would be written over the handle of the instance being deleted, and the number of items in the list would be decremented by one. The instance index field 874 for an instance object 871 may be used to facilitate fast deletions.

In a preferred embodiment, the value of parameters are always stored in base units. The objects in fields described do not necessarily occupy a word of memory. In a preferred embodiment, all parameters of a particular type are stored contiguously. This improves the speed of searches. For example, the case type 841' described with reference to **Figure 51** would be stored contiguously with all the other parameters for case type. The numeric parameter of 5.0 volts would be stored in a different physical location in memory contiguous with other numeric volt parameters.

As described above, providing a class object structure 800 with a field 808 providing the subtree instance count for that class allows the system to virtually instantly display a parts count 172 to provide the user instantaneous feedback during the tree traversal steps of the users search. The process of finding a part essentially amounts to discarding the thousands of parts that do not have the attributes desired and narrowing the search down to a small number that do.

This is accomplished by navigating to the correct class from the root of the classification hierarchy. During this phase, the parts found indication 172 can be updated using the data structure field 808 indicating the subtree instance count. This provides significant response time advantages compared to actually counting the available instances at each step. The user has immediate feedback indicating the number of parts available in the selected tree. The combination of providing an object oriented hierarchical tree structure together with search criteria based upon any desired combination of attributes, while providing instantaneous feedback on the number of instances corresponding to the current search criteria and class provides significant advantages over data base management schemes that have been attempted in the past.

An important function of the dynamic class manager 134 is the ability to modify the database structure during operation. The database structure is known as the schema. The schema of the object oriented database is structured using classes. The classes contain attributes. The attributes may contain enumerators, and unit families. The ability to add, move and delete these items is important to the dynamic operation of the

database.

To add a class to the schema, three items must be known: the class name, the parent of the new class, and the location within the list of subclasses to insert the new class. **Figure 65** illustrates this operation. The first step 1840 converts the handle of the parent class into an actual class pointer. The parent pointer must be immediately tested in step 1841 prior to its use. If the pointer proves to be invalid, then the operation terminates at step 1842. If the pointer is valid, the insertion index is tested in step 1843. If it proves to be invalid, the operation is terminated in step 1844. Finally, the name of the class must be tested in step 1845 to determine if it fits the guidelines of valid class names. If the class name fails, then the operation terminates in step 1846. When step 1845 accepts the class name, the new class can be created. A new handle is created in step 1847 first, and then the new class is created in internal memory in step 1848. The new handle is inserted into the table of class handles in step 1849 of **Figure 66**, followed by the handle being added to the parents list of subclass handles in step 1850. The last operation is to cause the file manager 140to add the new class to the indicated parent on the secondary storage device 103.

To add an attribute to a class, three items must be known: the class handle of the owning class, the location in which to insert the new attribute, and the name of the attribute. **Figure 67** illustrates the adding of attributes. The first step 1930 is to convert the class handle into a class pointer, followed by the testing of that class pointer in 1931 to determine if it is a valid class pointer. If not, the procedure terminates in 1932. If the class pointer is determined to be valid, then the insertion index is validated in 1933. If the index fails the validation test, then the procedure terminates in 1934. If the validation of the index succeeds, then the operation continues in 1935 where the name of the attribute is tested. If the attribute name fails, then the operation terminates in 1936. If the name of an enumerated attribute is accepted in 1935, then the attribute can be created. Step 1937 creates a new handle for the attribute. Then the new attribute is created in step 1938. The new attribute handle is then added to the list of attributes

local to the owning class in 1939. The last step is 1940 of **Figure 68** to cause the file manager 140 to update secondary storage 103 with the new attribute. The operation is complete in step 1941.

The addition of an instance is shown in **Figure 69**. Adding an instance requires a class handle. The class handle must be converted into a class pointer in 1918. The class pointer is tested in 1919 to determine if it is a valid class pointer. If it is not valid, then the procedure terminates in 1920. If the class pointer is determined to be valid, then the procedure continues in 1921 with the generation of a new instance handle and a new instance object. The handle for the new instance is inserted into the handle table in 1922. The instance is added to the parents list of instances in 1923. The subtree instance count is incremented to reflect the presence of the new instance in 1924. The instance has now been created in memory, and needs to be added to secondary storage 103, which is done in step 1925 of **Figure 70**. The procedure is complete in step 1926.

The deletion of a class is shown in **Figure 71**. To remove a class from the database structure, the current class handle must be identified. The class handle is first decoded into a class pointer in step 2600. The class pointer is then checked to determine if it is a valid class pointer in 2601. If the class pointer is invalid, the procedure is terminated in 2602. If the class pointer is valid, then it is checked to determine if it is the root class in 2603. If the class pointer represents the root class, then the procedure terminates in 2604, because the root class cannot be deleted. If the class pointer does not represent the root class, it is checked to determine if the class represents a leaf class in 2605. If the class pointer does not represent a leaf class, the procedure terminates in 2604. If the class pointer is found to point to a leaf class, then operation continues in 2906 where all of the instances of this class are deleted. The process of deleting instances is described below with reference to **Figure 75**. In step 2607 all of the attributes which are local to the class being deleted are deleted. In **Figure 72** The class is then unlinked from its parent class in step 2608. The system checks to determine if the unlink was successful, and that the data structures which contain the class

list are intact in 2609. If the unlink failed, then operation stops in 2610. If the unlink succeeded, then operation continues in 2611 where the class object is actually deleted. In step 2612, the file manager 140 is instructed to remove the class object from secondary storage 103, and the operation completes in step 2613.

The deletion of an attribute is shown in **Figure 73**. To remove an attribute, the attribute handle must be decoded into an attribute pointer in step 1860. Step 1861 checks to see if the attribute pointer obtained from step 1860 is valid. If the attribute pointer is invalid, the procedure stops in 1862. If the attribute pointer is valid, the procedure continues in step 1863 by searching the entire subtree for all of the parameters in all of the subtree instances that are derived from this attribute. After searching, in step 1864 the system determines how many parameters were derived from this attribute. If there were parameters derived from this attribute, the operation proceeds to 1865, where the parameters are undefined. If there were no parameters derived from this attribute, then the procedure continues to step 1866. Likewise, after the parameters have been undefined in 1865, the operation continues to 1866. In step 1866, the attribute is unlinked from the defining class. In 1867 the procedure checks to determine if the unlink operation succeeded. If the unlink failed, then the procedure stops at 1868. If the unlink was successful, then the attribute object is deleted in 1869 in **Figure 74**. The file manager 140 is then instructed to remove the attribute from secondary storage 103 in step 1870. The operation is complete in step 1871.

The deletion of an instance is shown in **Figure 75**. An instance is deleted from the database by first converting the instance handle in step 2000 to an instance pointer. The instance pointer is checked to determine that it is indeed a valid instance pointer in 2001. If the instance pointer is invalid then the operation terminates in 2002. If the instance pointer is valid, then the instance is unlinked from its owning class in 2003. The instance object is itself deleted in 2004. The subtree instance counts is then decremented to indicate that one instance has been deleted from the subtree in 2005. The file manager 140 is then instructed to update the secondary storage

103 to reflect the deletion of the instance in 2006. The operation is complete in step 2007.

In **Figure 76**, moving a subtree to a new position in the class hierarchy is described. In step 1800, the move subtree procedure is called with a class to move, the destination parent class, and the position among its sibling classes at the destination specified. In step 1801, the class pointers for the class to be moved and the destination parent class are obtained. If the test for all valid pointers in step 1802 fails, step 1804 returns an error, else test 1805 is made to prevent the class from being trivially moved to its own parent. Step 1806 insures that the position among the subclasses of the destination parent class is within a valid range, with an error returned by step 1804 upon error. In step 1807, the class hierarchy above both the class to be moved and the destination class is analyzed to identify the nearest common ancestor class.

In step 1808 of **Figure 77**, the common ancestor is tested to see if it is identical to the class being moved. If it is, given that a test has already been performed to insure that the class is not being moved to its parent, then this is determined to be an attempt to move a class to a subclass of itself, and an error is returned. All other moves are legal, so the class is unhooked from its parent class in step 1809 and added to the list of subclasses for the destination class in step 1810. The destination class subtree instance count is incremented by the number of instances in the moved class in step 1811, and the subtree count of the original parent class of the moved class is decremented by the moved class instance count in step 1812. In step 1813 the permanent image of the knowledge base is updated through the file manager 140, with step 1814 returning successfully to the caller.

Figure 78 describes unhooking the moved class from its original parent class. In step 1815 the class pointer for the parent is obtained and used in step 1816 to get a list of subclasses for the parent class. If the class handle of the class to be moved is not in the resulting subclass list as tested in step 1817, the knowledge base is internally inconsistent and an error is returned to the caller, else the class is deleted from the parent class subclass list in step 1818 before a

successful return in step 1819.

Figure 79 describes the process of finding the nearest common ancestor of the class to be moved and the destination class. In step 1820, a temporary class handle is set to the handle of the class to be moved. Step 1821 gets the parent of the temporary class, initiating a loop that creates a list of classes in order from the class to move to the root. Each class encountered is added to a list in step 1822, with iteration being terminated if step 1823 shows that the root has been encountered. If the test in step 1823 fails, the temporary class handle is set to the handle of its parent class in step 1824 and iteration continues.

A similar list is created for the destination class in steps 1831 through 1828, moving to **Figure 80**. In step 1831, a temporary class handle is set to the handle of the destination class. Step 1832 gets the parent of the temporary class, initiating a loop that creates a list of classes in order from the class to move to the root. Each class encountered is added to a list in step 1826, with iteration being terminated if step 1827 shows that the root has been encountered. If the test in step 1827 fails, the temporary class handle is set to the handle of its parent class in step 1828 and iteration continues.

The final step 1829 iterates through the two resulting lists until a matching class handle is found. This is the handle of the nearest common ancestor, which is returned in step 1830.

2. Connection Manager

The connection manager 135 is a subsystem of the knowledge base server 132 that manages information about the current client connections. The connection manager 135 is responsible for creating, maintaining, and closing client 130, 133, or 144 connections to the knowledge base server 132. The connection manager 135 will create an instance of query manager 136 for each client 130, 133 or 144 connection. The connection manager 135 maintains a linked list of entries about these client connections. A graphical representation of the data maintained by the connection manager is shown in **Figure 81**.

Referring to **Figure 81**, the connection manager 135 maintains a connection list pointer 1070 for each connection which points to a connection list 1077. The connection list 1077 includes

data concerning the start time, time of last request, and time of last message 1071 for a client 130, 133 or 144. A total count 1072 for calls to the API 143 is maintained. A pointer to remote procedure call connection information 1073 is also maintained. A pointer to information concerning the associated database manager 139 is also maintained. The connection manager 135 also retains a read-only flag 1075 to control access, and a pointer is maintained to the associated query manager 136.

3. Query Manager

The query manager 136 is a subsystem of the knowledge base server 132 that interacts with the dynamic class manager 134 to provide query operations on the knowledge base 123. The query manager 136 is responsible for managing the query data structures, matching selectors to parameters, and building and managing lists of instances or classes that matched the query.

The following discussion references the data structures described in **Figures 158-163**. When the query manager 136 is instantiated, a query manager class 700 is created. Each instance of this class 700 contains a query handle manager for queries 711, a query handle manager for query results 712, and a query handle manager for search results 713. In general a query handle manager class 701 is a list of base query classes 702. This list 701 is the mapping between a handle and a base query class 702 or one of the derived classes, query class 703 , search result class 704, and query result class 705. The offset into the list represents the handle of the object.

A "query" is an object created through the API 143 that can be used to select instances in the knowledge base 123 based on parametric criterion. A query is always tied to a class when it is created.

To create a query, query class 703 is created as a derived class of base query class 702. There is one of these classes 703 for every query created. The base query class 702 is the base class for queries (query class 703), query results (query result class 705), and search results (search result class 704). Base query 702 contains the query class handle 714 that is the class handle of the class on which a query was created. Since the query manager 136 needs to access the dynamic class manager 134, a reference 715 to the dynamic class manager 134 is kept.

Once created, a query class 702 continues to exist until it is explicitly deleted.

A query consists of zero or more "selectors". A "query selector" is associated with one of the attributes defined for the class for which the query was created. Setting a selector will cause the query to return only those instances that match the selector. Setting multiple selectors causes the query to return the conjunction of the instances matching the selectors. i.e., only instances matching all selectors will be returned

The exact form and semantics of the query attribute class 706 selector depends on the type of the associated attribute. Each attribute can have at most one associated selector class 706 or derived classes 707, 708, 709, or 710 for any given query. For any attribute type, the selector class 706 or derived classes 707, 708, 709, or 710 can be set to include instances for which the associated parameter is in the "undefined" state 731. Setting a selector class 706 without requesting the undefined parameters will cause the instances that have that parameter set to undefined to be excluded. Requesting that undefined parameters be included without otherwise setting the selector class 706 will cause only the instances for which the parameter is undefined to be returned.

The attribute selector classes 707, 708, 709, and 710 are added to the query class 702 attribute selector class list 716 as a result of an API 143 call to set the specified attribute selector class 706.

The attribute selector class list 716 is destroyed when the query class 702 is destroyed.

Actually performing a query is referred to as "applying a query." Applying a query will return a query result handle. A query result handle is a reference to a query result class 705. A query result class 705 is an object that contains the list of instances 723 returned by the query. Given a query result handle , the user can retrieve the actual instances represented by list 723. The query result continues to exist until explicitly deleted. Subsequent changes to the query class 702 will not affect an existing query result class 705. Subsequent applications of the query class 702 will return additional query results class 705.

The query may be applied either locally or on a subtree. When a query is applied locally, the query manager 136 retrieves the class pointer associated with the class handle 714 using the class manager 134 reference 715. The list of instances associated with the class pointer is retrieved and a list iterator is used to evaluate each instance against the attribute selector list 716 in the query class 703. A query class 703 with no selectors will simply return all instances of class 714.

In **Figure 82** and **Figure 83** a local query is applied. The query handle is converted to a query class 702 pointer in step 750. In step 751, the pointer is validated. If the pointer is invalid, an error is returned in step 752.

In step 753 , the query result class 705 is created and added to the query result handle manager 712.

In step 754 , the class manager 134 is called using reference 715 to get the class pointer for the class handle 714 that was set in the query class 702. This pointer is used in step 755 to call the class manager 134 function to get the list of instances for the class handle in 714 .

The query class 702 class pointer from step 750 is used to retrieve the selector class list 716 associated with the query class 702 class in step 756.

A significant performance optimization of this invention is checking in step 757 to see if any selector classes 706 and derived classes are set. If no selector classes 706 and derived classes are set, step 760 is performed which associates the class instance list with the query result class 705 and a normal return is done in step 761.

If selector classes 706 and derived classes are set, then step 758 requires an instance in the class instance list to be examined. If no more instances are available in step 759 , the instance list of matches is associated with the query result class 705 in step 760 in list 723 and a normal return is done in step 731.

A selector class 706 and derived classes are retrieved from the query class 702 attribute selector class list 716 in step 762 . In step 763 , if there are no more selector classes 706 and derived classes to evaluate, the process returns to step 758 to get the next instance in the class after saving the

instance handle in the query result class 705 instance list 716 in step 769.

If there are selector classes 706 and one of its derived classes to evaluate, step 764 is performed. This step retrieves the parameter value for the attribute handle 763.

Another significant performance optimization of this invention is the process that starts in step 765 to check if the parameter is defined. This step makes processing of empty or null values extremely efficient. If the parameter is not defined, the undefined selector flag 731 is checked in step 768. If the undefined flag 731 is not set, the instance handle is discarded as a possible match and the next instance is processed starting in step 758 . If the undefined flag 731 is set, the instance matches and the next selector list 716 item is processed in step 762.

If parameter values existed in step 765, the selector list 716 item is checked in step 766 to see if criteria are set. If they are not, the next instance is processed in step 758. Based on the attribute type, the selector 706 and derived classes are used for evaluating the parameter and selection criteria in step 767. If the selection criteria matched, the next selector class list 716 item is processed in step 762.

In **Figure 84**, the process for performing a query on a subtree is shown. The query handle is converted to a query class 702 pointer in step 770. In 771, the pointer is validated. If the pointer is invalid, an error is returned in step 772.

In step 773, the query result class 705 is created and added to the query handle manager 712. The next step 774 performs the apply local query function described in **Figure 82** and **Figure 83**. This step 774 will be applied recursively for each subclass of the class handle 714. The subclasses for this class handle 714 are retrieved from the class manager 134 reference 715 in step 775.

If there are more subclasses to process as determined in step 776, the class handle for the subclass is retrieved from the class manager 134 reference 715 in step 781 and the local query procedure in **Figure 82** and **Figure 83** is called in step 774.

From step 776, if all the subclasses were processed, the

procedure returns to the parent class in step 777. Step 778 checks to see if the recursive algorithm has returned back to the top query class 714 . If it hasn't, the remaining subclasses of the current class are processed in step 776. If the procedure has returned to the top query class 714, the instance list 723 is associated with the query result class 705 in step 779 and the procedure returns to the caller in step 780.

The data structures that are created when a query is created and applied are shown in **Figures 164**, **Figure 157**, **Figure 156**, and **Figure 155**. In these figures assume the root class of a particular knowledge base has a boolean attribute "discontinued" and a numeric attribute "length" with a base unit of "inches". The user does a query of all discontinued parts with a length >3 inches.

In **Figure 164** a query on the root class (class handle 0) is created using the API 143 function "pmx_createquery". A query handle of 2 is returned to the caller. The user then retrieves the handles for the boolean attribute "discontinued" (attribute handle 10) and for the numeric attribute "length" (attribute handle 19 and unit handle 5) using API 143 function get "pmx_getattributedescriptorset".

In **Figure 157**, the results of setting the boolean selector using API 143 function "pmx_setbooleanselector" with queryhandle 2 is shown.

In **Figure 156**, the results of setting the numeric selector using API 143 function "pmx_setnumericselector" with queryhandle 2 is depicted.

After applying the query handle 2, instances 3, 300, and 30000 are found. The results of applying the query are shown in **Figure 155**. A query results handle of 0 is returned since there are no other query results in this example.

Another significant performance optimization of this invention is described in **Figure 85** applying a query count. This procedure is used to return quickly and efficiently the number of parts available in a schema class tree to the retriever 130. The process starts in step 790 when the query class 702 pointer is converted from the query handle. Step 791 checks this pointer for validity, and returns in step 792 if an error occurred. Step 793 accesses the dynamic class manager 134 using reference 715

to get the class pointer for the query class 714. The list of selectors 716 described by the base attribute selector class 706 is retrieved in step 794. If a selection list 716 item exists as determined in step 795, the procedure for applying a query described in **Figure 84** must be executed and the resulting instance count returned in step 798. The significant invention occurs in step 796. The dynamic class manager 134 directly maintains a count of instances local to a class, as well as a count of all instances in the subtree parented by that class. This value is maintained when instances are moved, deleted, or added to a class. When the apply query count procedure is applied, the value is simply looked up in the class and returned to the user in 796. This step results in high performance tree traversal feedback in the retriever 130, item 172.

The initial order of instances within a query result 705 is random. The query result instances 723 can be reordered using an API 143 function for sorting the instances. The retrieval functions will then return the instances in sorted order. A query result class 705 can be resorted multiple times.

In **Figure 161**, a sorting request consists of an ordered list of attributes 719 and an indication of whether the order is to be ascending or descending in sort order list 720. The descending order is precisely the reverse of the normal order which is ascending order.

The instances will be ordered first by the first attribute in the ordering list 719. Within a group of instances that have equal settings for the first attribute, the second attribute will be used, and so on until the list is exhausted. Any order not uniquely determined by the list will be essentially random.

The ascending order for boolean attributes is (TRUE, FALSE, UNDEFINED). The ascending order for enumerated attributes is the order of the enumerators as defined in the schema followed by undefined. The ascending order for string attributes is the normal ASCII collating sequence followed by undefined.
The ascending order for numeric attributes is first sorted by base units in the order defined by the schema. Within a base unit, the instances are in numeric sequence. Undefined parameters will be last.

Additional file manager 140 derivations are possible. The

interface provided by the file manager 140 to the dynamic class manager 134 and the handle manager 137 is an agreement to maintain a copy of the dynamic class manager schema and instance data on secondary persistent storage 103. Changes, as they are made to the schema and instances are also made in secondary storage. The dynamic class manager 134 is initialized by reading the data, via the file manager 140, from secondary storage 103. Other secondary storage mechanisms could be implemented which follow the interface specification. Other implementations could use commercial data bases including relational database management systems such as an Informix database, Oracle database, Raima database, etc. Other implementations could also be built using other proprietary file formats.

A query result is instantiated and populated with instance handles. The order of the instances is random within the query result. Instances within a query result may be sorted according to specified criteria. Multiple sort criteria may be specified. Sort criteria is specified by indicating a list of one or more attributes and a sort order (ascending or descending) for each attribute. Instances in a query result are ordered by sorting the parameters indicated by the attributes according to the specified order.

A query result is not sorted immediately upon receiving sort message containing the sort criteria. Rather a query result merely remembers the sort criteria and waits to perform the sort when an instance handle is requested from a query result. (From the PMX Retriever, a request for an instance handle is eminent.)

When an instance handle is requested from a query result, the query result will sort itself at this point. However, the entire query result is not sorted; only the portion of the query result containing the instance of interest is sorted. The sorting method used is incremental, only sorting those portions of the query result that contain instances of interest. The other portions of the query result are left unsorted. Incremental sorting is done to improve response time: sorting a portion of the query result usually takes less time than sorting the entire query result.

Incremental sorting requires tracking which instance handles in a query result have been sorted and which have not. To

accomplish this, the query result is sub-divided in to ranges. There are two types of ranges: sorted and unsorted. Every instance handle in a query result resides in either a sorted or unsorted range. The Range Manager is responsible for managing these ranges.

Initially, prior to any sorting, the entire query result is contained in a single unsorted range. The act of incrementally sorting the query result will sub-divide the query result into several ranges, some which are sorted, some which are not. As more and more portions of the query result are sorted, the number of unsorted ranges become fewer; eventually the entire query result becomes a single sorted range.

The range manager is aware of the meaning of sorted and unsorted ranges and uses this information to join ranges together to avoid range fragmentation, which would decrease performance speed. The range manager uses two rules to join ranges: 1) two adjacent ranges of the same type (sorted or unsorted) may be joined together to make a single larger range, 2) an unsorted range containing only a single instance handle may be joined with an adjacent sorted range to make a larger sorted range.

Prior to sorting, a query result consists mainly of a list of instances, 0 through N, in an unsorted state step 1350 of **Figure 144**. The receipt of a sort message, changes the state of the query result to sorted and provides it sorting criteria step 1351 of **Figure 1444.**

The query result will eventually receive a message to return the Ith instance handle from the list of instance handles in step 1352 of **Figure 188**. At this point the query result must sort the list to correctly order the Ith instance. This is done by first selecting a random index, R, between 0 (lower bound of range) and N (upper bound of range) (step 1353). The entire list is ordered such that all instances greater than the instance at R are placed above R in the list and all instances less than the instance at R are placed below R in the list (step 1354). The instance at R is now ordinally sorted within the list. At this point there are three ranges: 0 to R-1 (unsorted), R (sorted), and R+1 to N (unsorted).

If Ith and R coincide, sorting is complete and the Ith instance handle is returned. Otherwise sorting must continue.

Sorting continues based on the location of Ith in relation to R. If Ith is less than R, then the range between 0 (lower bound) and R-1 (upper bound) will be ordered. If Ith is greater than R, then the range between R+1 (lower bound) and N (upper bound) will be ordered. With the appropriate range determined, a new random R is selected within the new range, and all instances in the new range will be partially ordered with respect to the instance at R.

The list is iterively sub-divided into ranges and partially ordered until Ith and R coincide, at which time the sorting discontinues and Ith instance handle is returned.

Sometime later when a another Ith instance is requested, if this Ith instance is found within a range that has already been sorted, no sorted need be done, and the Ith instance handle is immediately returned. If this Ith instance is not found within a sorted range, the sorting continues by iterively ordering the ranges until Ith is found.

Each time a range is ordered and new ranges are identified, the range manger is provided with this information. The range manager keeps track of all ranges. At beginning of each iteration range manager is asked if Ith exists in sorted range. If it is, then no further sorting is required. If Ith is not found within a sorted range, then the range manager provides the lower and upper bound of the range that Ith exists in and the algorithm orders that range and creates new ranges.

At the end of a sorting session, just prior to returning the Ith instance, the range manager is provided the opportunity to join fragmented ranges based on the rules previously mentioned. At the end of a sorting session, the range manager may be tracking the step shown in 1355 of **Figure 189**. Any adjacent ranges of the same type may be joined together. Also, unsorted ranges of size one, may be joined to adjacent sorted ranges.

The range manager will be tracking the ranges shown in 1356 when the range joining is complete.

A class pattern search is similar to a query but is applied to all of the classes in a subtree. A search result is represented by search result class 704.

A class pattern search performs a pattern match on the class names and returns a list of classes 717 whose names match the

pattern. Unlike a query, there is no separation of creating a search and performing it, since a search isn't complicated enough to need to be built up a piece at a time. The search returns a "search result" handle which is an item that continues to exist until the user deallocates it. There is no defined order in which the classes are returned.

4. Handle Manager

The handle manager 137 is a component of the dynamic class manager 134 that provides services for creation, deletion, and disk-to-memory mapping of handles for all objects. The handle manager 137 comprises two lists of virtual memory addresses which are shown in **Figure 42**. The first list 810 contains the virtual memory addresses 810-814 of schema objects (classes, attributes, enumerators, units, and unit families). The second list 811 contains the virtual memory addresses 815-826 of instances. A handle is an index into a list. Thus, given a schema object handle or an instance handle, the handle manager 137 can return to the dynamic class manager 134 the virtual memory address of the desired object.

When the dynamic class manager 134 needs to examine the data for some object for which it has a handle, the handle manager 137 responds to a request for the virtual memory address of the object as shown in **Figure 52**. The procedure begins at step 1000 with a request from the dynamic class manager 134. The handle is checked for validity at step 1001 (i.e., that the handle is one that was created by the handle manager 137). If the handle is not valid, an error condition is generated and the handle manager returns a NULL virtual memory address to the dynamic class manager 134 to indicate the error in step 1002. Otherwise, the handle manager 137 continues with step 1003.

If the handle is valid, then the address stored in the appropriate list (schema object or instance) is examined at step 1003. One special virtual memory address is reserved to indicate that an object with the given handle is deleted. Only objects which are deleted are allowed to have this special memory address. If the address found from the handle look up in step 1003 is the deleted object address, then an error condition is generated and the handle manager 137 returns a NULL virtual memory address in step 1004 to the dynamic class manager 134.

Otherwise, the handle manager 137 continues with step 1005. If the virtual memory address found in the list at step 1005 is not a NULL pointer, then processing continues at step 1009. If the virtual memory address found at step 1005 is NULL, then the requested object is not present in memory. The handle manager 137 makes a request to the file manager 140 to read the object with the given handle from secondary storage 103, create the object in the virtual address space, and return the virtual memory address to the handle manager 137 in step 1006.

At step 1007, the virtual memory address of the object which has been created by the file manager 140 is tested against the special deleted virtual memory address. If file manager 140 has determined that the object is deleted, then an error condition is generated and a NULL pointer is returned in step 1008. Otherwise, processing continues at step 1009.

At step 1009, the handle manager 137 has identified a valid virtual memory address for the object with the given handle. The type of the object is tested to insure it is of the same type as the type of the handle. If the type is not correct, then an error condition is generated and a NULL pointer is returned in step 1010. Otherwise, the address of the requested object has been identified and this address is returned in step 1011 to the dynamic class manager 134.

When the dynamic class manager 134 creates a new schema object or instance on behalf of a function of the API 143, the handle manager 137 is invoked to generate a new handle for the new object. The handle manager 137 returns an unused handle, which is the next list index sequentially following the most previously generated handle. In other words, it returns oldmax + 1. The handle manager 137 is informed of the address of the new object so it can be entered into the list.

The handle manager 137 is also invoked by the dynamic class manager 134 whenever an object is deleted on behalf of a function of the API 143. The virtual memory address stored in the list which is indexed by the given handle is set to the special deleted object address.

5. Units Manager

The units manager 138 is an integral component of the dynamic class manager 134 that provides services for creation,

maintenance, and deletion of base and derived units for numeric attributes. The data structures provided are discussed in detail as part of the description of the dynamic class manager 154. The presence of the units manager 138 and its ability to relate numeric quantities to the units in which they are measured and to perform automatic conversions among compatible units when updating, searching and sorting the numeric values in the database 123 has significant advantages compared to storing the numeric values devoid of units.

6. File Manager

The file manager 140 is a subsystem of the knowledge base server 132 that provides access to a secondary storage mechanism 103 for the schema objects and instances. The file manager 140 provides an access method independent set of services for reading, writing, updating, creating, and locking knowledge bases 123.

The file manager 140 provides to the dynamic class manager 134 and handle manager 137 an abstract interface to the persistent storage 103 of knowledge base objects. In other words, the file manager 140 is a C++ abstract base class which is intended to be fully defined by derived classes. The interface functions or methods provided by the file manager 140 are shown in **Table 5** and **Table 6.** The functions provided by the file manager 140 may be separated into groups depending on their usage.

Functions for opening and closing secondary storage are used by the class manager 134 when the class manager 134 is created to service a knowledge base 123when a knowledge base server 132 is started or when the knowledge base server 132 terminates. The class manager 134 uses a warm start function to initialize the knowledge base server 132 in the desired configuration. A factory creation function is used by a file manager factory. Those skilled in the art are familiar with the use of factories for object instantiations, and such functions will not be described in detail. See Coplien, Advanced C++.

Other file manager 140 functions are used by the dynamic class manager 134 whenever it performs some operation which modifies the knowledge base 123. These functions correspond nearly one-to-one for the API 134 and dynamic class manager 134

functions which make modifications to the knowledge base 123. The file manager 140 is responsible for insuring that the data in secondary storage 103 models exactly the data in the dynamic class manager 134.

Additional file manager functions are used by the handle manager 137 when the dynamic class manager 134 uses the handle of some object which is not in virtual memory (see **Figure 52**, step 1006). These functions construct the object in virtual memory by reading the object from secondary storage 103. The address of the created object is returned to the handle manager 137.

Table 5

Functions used by the class manager for API actions:

```
virtual long getDBFeatureCode
virtualconst cd_stringList CD_FAR & getDBCopyright
virtual cd_boolean addLeafClass
virtual cd_boolean addInstance
virtual cd_boolean addAttribute
virtual cd_boolean addEnumerator
virtual cd_boolean addStringArrayElement
virtual cd_boolean changeSubclassOrder
virtual cd_boolean changeAttributeOrder
virtual cd_boolean changeEnumeratorOrder
virtual cd_boolean deleteLeafClass
virtual cd_boolean deleteInstance
virtual cd_boolean deleteAttribute
virtual cd_boolean deleteEnumerator
virtual cd_boolean deleteStringArrayElement
virtual cd_boolean moveInstance
virtual cd_boolean moveAttribute
virtual cd_boolean moveSubtree
virtual cd_boolean setParameter
virtual cd_boolean setStringArrayElement
virtual cd_boolean setParameterUndefined
virtual cd_boolean setClassName
virtual cd_boolean setAttributeName
virtual cd_boolean setEnumeratorName
virtual cd_boolean setClassCode
virtual cd_boolean setAttributeRequired
virtual cd_boolean setAttributeProtected
virtual cd_boolean addUnitFamily
```

```
virtual cd_boolean addUnit
virtual cd_boolean setEnumeratedUnitRows
virtual cd_boolean setClassMetaParameters
virtual cd_boolean setAttributeMetaParameters
virtual cd_boolean setEnumeratorMetaParameters
virtual cd_boolean setUnitMetaParameters
virtual cd_boolean setEnumeratedUnitTable
virtual cd_boolean setUnitFamilyName
virtual cd_boolean setUnitName
virtual cd_boolean setUnitConversionValues
virtual cd_boolean deleteDerivedUnit
virtual cd_boolean setAttributeUnits
```

Table 6

Functions for opening and closing secondary storage:

```
cd_fileManager
virtual ~cd_fileManager
```

A function for warm starting:

```
virtual cd_class CD_FAR * warmStart
```

A function for factory creation of a derived file manager:

```
static cd_fileManager * make
```

Functions used by the handle manager for faulting objects into memory:

```
virtual void getClass
virtual void getAttribute
virtual void getEnumerator
virtual void getUnit
virtual void getUnitFamily
virtual void getSchemaObject
virtual void getInstance
```

The presently preferred embodiment comprises two derivations of file managers 140 from a base file manager class. A null file manager 140 (called **"nullmgr.hxx"**) defines all of the file manager 140 functions, but the effect of any of these functions is null. The null file manager 140 provides no secondary storage for the dynamic class manager 134. The purpose for this type of

file manager 140 is primarily for testing.

A second derivation of the file manager 140 is the Cadis File Manager (called **"cdsdbmgr.hxx"**). The Cadis File Manager interacts with secondary storage for persistent storage of the schema objects and instance objects. The formats of the files as stored on secondary storage are shown in **Figures 53 - 64**. The Cadis File Manager also manages the details of the mapped I/O, the standard I/O, and the raw I/O access methods.

The Cadis File Manager maintains a copy of the current state of the knowledge base 123 in simple files on secondary storage 103. Although the secondary storage copy can be thought of as a single knowledge base 123, for convenience it is mapped to three files on secondary storage. These three files are known as the schema file, the instance file and the dynamic file 2400. The schema file and instance file contain fixed size data about schema objects and instance objects respectively. The dynamic file contains data, such as lists of items and character strings, which by their nature are of varying length.

Referring to **Figure 53**, the sequential layout of the dynamic file 2400 is shown. The dynamic file 2400 contains a header 2401 (described below) and, following sequentially, a plurality of variable length objects. The first variable length object is 2402, the second 2403. These objects continue down through the file. **Figure 54** shows the general layout of the schema and instance files 2404. Here the format is essentially the same with a header 2401 and a series of objects 2405, 2406 etc. In each of the schema and instance files, however, the objects are of known size. This means that they can be located quickly in the file knowing only their ordinal position amongst the objects. In the current implementation, this ordinal position is always exactly equal to the value of the handle assigned to the object.

Figure 55 shows the layout of the file header 2401 which is present in all three files. The first six computer storage words in the headers of the three files follow the same format across files. These six words contain the release number 2407 and revision number 2408 of the knowledge base 123, the date 2409 when the knowledge base was created, the file offset 2410 of the last location in the file which contains data, a boolean flag 2411 indicating whether the knowledge base 123 can be updated,

a feature code 2412 optionally indicating the source of the data present in the file, and two filler words 2413 and 2414. The headers of all three file types will then contain two additional words. The contents of these words will vary among the files. The schema file 2404 will contain an offset into the dynamic file 2400 where a list of global units is present 2415 and the value of the maximum handle used in the schema file 2416. The instance file will contain an additional filler word 2417 and the value of the maximum handle used in the instance file 2418. The dynamic file will contain an additional filler word 2419 and a word containing the value "-1" 2420.

The objects 2405, 2406 etc. present in the schema file will be objects of various types corresponding to the various types in the schema. In most cases, the values stored in the knowledge base for a particular type of object correspond in a very straightforward fashion with the values kept in memory by the dynamic class manager 134. **Figures 56, 57, 58, 59** and **60** show the layouts of these various objects. Each of these object types is comprised of twelve computer storage words.

Figure 56 shows the layout of a schema file object 2421 which represents a class in the knowledge base 123. The class object contains a flag indicating if the class has been deleted 2426, a type code which is always "20" 2427, an indicator of whether the class is a "primary", "secondary" or "collection" class 2428, an empty filler byte 2429, the handle of the class 2430, the handle of its parent class 2431, an offset into the dynamic file where the list of subclasses belonging to the class can be found 2432, an offset into the dynamic file where the list of local attributes of the class can be found 2433, an offset into the dynamic file where the list of instances belonging to the class can be found 2434, the number of instances currently located in the subtree rooted at the class 2435, an offset into the dynamic file where the list of metaparameters which belong to the class can be found 2436, three filler words 2437, 2438 and 2439 and an offset into the dynamic file where the name of the class can be found 2440.

Figure 57 shows the layout of a schema file object 2422 which represents an attribute in the knowledge base. The attribute object contains a flag indicating if the object has

been deleted 2441, a type code 2442 which is 51 for an enumerated attribute, 52 for a boolean attribute, 53 for a numeric attribute, 54 for a string attribute and 55 for a string array attribute. It also contains a field indicating if the attribute is "required" 2443, a field indicating if it is "protected" 2444, the handle of the attribute 2445, and the handle of the class which defines the attribute 2446. If the attribute is an enumerated attribute, there will be an offset into the dynamic file where the list of enumerator handles belonging to the attribute will be found 2448. If the attribute is a numeric attribute, there will be the unit family handle 2449 for the unit family which contains the units which the numeric attribute uses. If the attribute is not of one of these two types, a filler word 2447 will be present. The attribute will contain the offset into the dynamic file where the meta-parameters for this attribute are listed 2450 and filler words 2451, 2452, 2453, 2454, 2455 and 2456. Finally, the attribute will contain the offset into the dynamic file where the attributes name is given 2457.

Figure 58 shows the layout of a schema file object 2423 which represents an enumerator in the knowledge base 123. The enumerator object contains a flag indicating if the object has been deleted 2458, a type code 2459 which always contains the number "60", two filler bytes 2460, the handle of the enumerator 2461, the offset into the dynamic file where the meta-parameters for the enumerator can be found 2462, filler words 2463 through 2470 and the offset into the dynamic file where the name of the enumerator is located 2471.

Figure 59 shows the layout of a schema file object 2424 which represents a unit in the knowledge base. The unit object contains a flag indicating if the object has been deleted 2472, a type code 2473 which is "81" for a base unit, "91" for a real derived unit and "92" for an enumerated table, a unit type flag indicating whether the unit is integer, real or enumerated 2474, a field which, for an enumerated unit, contains the number of rows to be displayed in the table 2475, the unit's handle 2476, the handle of the unit family which defines this unit 2477 and the handle of the base unit from which this unit is derived 2478 (or NULL if this is unit is a base unit). For a base unit, there will then be two filler words 2479 and 2480. A real derived unit

has a multiplication factor 2481 and an offset 2482. An enumerated table has an offset into the dynamic file where the list of enumerator strings is located 2483 and an offset into the dynamic file where the list of real values is located 2484. All unit types then have an offset into the dynamic file where the meta-parameter list can be found 2485, four filler words 2486-2489 and an offset into the dynamic file for the unit name 2490.

Figure 60 shows the layout of a schema file object which represents a unit family in the knowledge base 2425. The unit family object contains a flag indicating if the object has been deleted 2491, a type code which is always "70", 2492, a two byte filler field 2493, the handle of the unit family 2494, the offset into the dynamic file where a list of unit family handles which are included in this unit family can be found 2495, and offset into the dynamic file where a list of unit handles defined by this family can be found 2496, seven filler words 2497-2503, and an offset into the dynamic file where the name of the unit family can be found 2504.

The objects 2405, 2406 etc. present in the instance file will all be instance objects. Each instance object is comprised of four computer storage words. **Figure 61** shows the layout of an instance file object 2511. The instance object contains a flag indicating if the instance has been deleted 2505, a type code which is always "30", a two-byte filler field 2507, the handle of the instance 2508, the handle of the class which owns the instance 2509, and an offset into the dynamic file where the list of parameters belonging to the instance can be found 2510.

The objects 2402, 2403 etc. present in the dynamic file are variable length objects which have various types based on the size of the components which are stored therein. **Figure 62** shows the layout of a type 1 dynamic object 2512 which is used to store a character string. A type 1 dynamic object contains a flag to indicated if it has been deleted 2516, a type code which is "1" 2517, the length of the character string stored 2518, the amount of space actually allocated in the file for the character string 2519, a two-byte filler 2520, and a block of characters which contain the stored string 2513. **Figure 63** shows the layout of a type 2 dynamic object 2514 which is used to store data items which are four bytes in length, such as handles, integers, reals,

offsets, etc. A type 2 dynamic object contains a flag to indicate if it has been deleted 2521, a type code which is "2" 2522, a two-byte filler 2523, the length of the stored data 2524, the amount of space actually allocated in the file for the data 2525, and a block of data which contains the actual stored values 2515. **Figure 64** shows the layout of a type 3 dynamic object 2526 which is used to store parameter data. Each stored parameter takes 4 computer words. A type 3 dynamic object contains a flag to indicated if the object has been deleted 2527, a type code which is "3" 2528, the length of the stored data 2529, the amount of space actually allocated for the data 2530, a two-byte filler 2531, and then a succession of parameter objects 2532, 2547 and so forth. Each parameter object 2532 contains a flag indicating if the parameter has been deleted 2533, a type code which indicates if the parameter is enumerated, Boolean, numeric, string or string array 2534, a two-byte filler 2535 and the attribute handle of the attribute to which this parameter refers 2536.

If the parameter is of enumerated type, the parameter object will also contain the handle of the enumerator to which the parameter is set 2537 and a filler word 2538.

If the parameter is of Boolean type, the parameter object will also contain the actual Boolean value stored 2539 and a filler word 2540. If the parameter is of numeric type, the parameter object will also contain the handle of unit in which the value is expressed 2541 and the actual numeric value 2542 expressed in those units. If the parameter is of string type, the parameter will also contain an offset into the dynamic file where the string value is located 2543 and a filler word 2544. If the parameter is of string array type, the parameter will contain an offset into the dynamic file where a list of offsets to the stored character strings can be found 2545 and a filler word 2546.

7. DataBase Manager

The database manager 139 is a subsystem of the knowledge base server 132 that stores and manages high-level information about knowledge bases 123 being managed by the knowledge base server 132. A graphical representation of the data maintained by the database manager 139 is shown in **Figure 152**. The database

manager 139 maintains a linked list of entries about knowledge bases 123 managed by the knowledge base server 132.

The database manager 139 is responsible for concurrency control on database objects. For concurrency control, write locks are maintained on classes. A write lock has the property that read or retrieval operations may be performed on the locked class, but update operations may only be performed by the lock holder. Locks are set by the schema editor 500 and by legacy 133 to allow concurrent updaters and privacy in legacy work areas.

Only one lock holder is allowed per class. Lock holders are identified by their connection, not by user name. Locks are maintained for the length of a connection. Once a connection is destroyed by either closing the knowledge base 123 or because the connection timed out, all locks held by that connection are released.

Locking a class locks all attributes defined by that class. Locks are required for modifying attributes and classes. Locks are advisory for editing instances.

The granularity of locking is at the knowledge base, tree and class level. Locks may be set locally to a class or inherited. Local class locks are set using a class lock mechanism. These are local locks which are not inherited by subclasses of the locked class. For example, the root class of a knowledge base 123 may be class locked to prevent updates, but the subclasses may still be locked by another user.

Locks may be inherited by locking the knowledge base 123, which implicitly locks all classes in the knowledge base 123. Locks may also be inherited by locking a subtree. A subtree is locked by applying a tree lock to a class. All descendent classes of the tree locked class are locked by implication. Physically, any class locks in the subtree are subsumed by the subtree or knowledge base lock. For a user to get a tree lock, no nodes in that tree can be locked by another user.

For a more detailed discussion of lock object granularity, see Won Kim, "Object Oriented Databases", or Won Kim, "Object-Oriented Concepts, Databases, and Applications", (1989) published by ACM Press.

In **Figure 86**, let class B be locked by User 1. User 1 could be granted a tree lock on class A since there are no locks held

in the tree by other users. In another example, let class B be locked by user 1. Locks can be granted to User 2 for classes C, D, E, and F since there are no other lock holders for those classes. User 2 can be given a local tree lock on class A, but a tree lock would be denied user 2 since class B is locked by user 1.

One feature of the invention is the ability to specify that an interface requires a lock on an attribute or class that is a parameter to the interface. The database manager 139 will check for a lock at the database manager level, and this relieves the class manager 134 from subsequent lock conflict resolution.

D. API

The application programming interface or API 143 refers to the external C or C++ language functions that provide access to the functions provided by the knowledge base server 132, registry server 141, and license manager 142 functions to client applications 130, 133, and 144.

E. Registry Server

The registry server 141 is a UNIX process that provides administration and security functions for users and knowledge bases. User administration functions include name and password management and mapping user access rights to knowledge bases 123. Knowledge base administration provided by the registry server includes RPC service mapping, host CPU mapping, and logical to physical name mapping.

F. License Manager

The license manager 142 is a UNIX server process (which in the illustrated example is called "pmxlm") that provides software license control for the software and for licensed knowledge bases 123. Satisfactory operation of the license manager 142 may be achieved using a conventional Elan License Manager available from Elan Computer Group, Inc.

G. Schema Editor

The schema editor 144 is an application that provides a graphical interface for creating, editing, and deleting schema objects. Objects may be renamed, reordered, and moved. The schema editor 144 communicates with the knowledge base client 131 using the API 143. The schema editor 144 provides an object oriented graphical user interface. A user interacts with schema

editor 144 providing input through a keyboard 115 and a mouse 114. The schema editor 144 displays information on the display 116.

Figure 87 depicts a typical display that appears on the screen of the display 116 after a user successfully logs on to the system and selects schema editor from the e tools pull down menu 146 from the parts specification window 170 shown in **Figure 88**. The particular example described herein is described in a Windows environment, it being understood that the invention is not limited to implementation in Windows. Those skilled in the art are familiar with Windows techniques and instructions, including how to click, double click, drag, point and select with a mouse 114. Additional information may be obtained from the Microsoft Window's User's Guide (1992), available from Microsoft Corporation, One Microsoft Way, Redmond, Washington, 98052-6399, part number 21669.

When a user first opens the schema editor 144, a schema editing window 500 appears, as shown in **Figure 89**. Initially, the left hand portion of the screen 501 displays the class title edit box 502, which is used to change the title of the selected class. The class title OK button 503 and cancel button 504 are used to accept or reject class title changes. The class add button 505 and delete button506 are used to add or delete classes. Also displayed on the left-hand portion of the screen 501 is the root class 507 and the root subclasses 508. In the illustrated examples, the root subclasses 508 are "electrical components", "mechanical", and "materials". The root class 507 is the upper most class that has no parent. In this example, it is the name of the knowledge base 123, or the very beginning of the schema. A subclass 508 is a class that has a parent. When a class 507 is chosen, any subclasses 508 that belong to that class 507 will appear on the display 501. Subclasses are the children of the parents. For example, the parent of the mechanical subclass 508 is the root class 507, and the mechanical subclass 508 is a child of the parent root class 507. In the example shown in **Figure 89**, there are three subclasses 508.

The right hand portion of the screen 509 displays the root attributes 516. In the illustrated example, the attributes are "part number", "description", and "cost". Attributes 516 are the

characteristics of a class or subclass 507. Attribute number column 517 is used to display the total attributes both local and inherited for the selected class represented on the class side of the screen 501. The locks column 519 and the required column 520 are used to set locked (protected) or required attributes. The user clicks on the row of the desired attribute in the lock column 519 or the required column 520, a check mark will appear in the selected row/column if the lock or required is turned on. Locked and required attributes are used for make part described above in connection with the description of the retriever 130. Also displayed in the right hand portion of the screen 509 is attribute title edit box 510, which is used to change the title of the selected attribute. The attribute title OK button 511 and cancel button 512 are used to accept or reject attribute title changes. The attribute add button 513, delete button 514 and edit button 515 are used to add, delete, or edit certain attributes. The command name in these buttons is dimmed when the user has selected an attribute that is not owned by the selected class in area 501. The edit button is also dimmed if the attribute type is not either numeric or enumerated.

Class tree 508 is navigated by double clicking on the closed folder icon 189 as described in the flow chart in **Figure 90** and in connection with **Figure 91**. The user double clicks on a closed folder icon 529 in step 521, an open folder is displayed and a list of subclasses is obtained in step 522. For each subclass that was obtained, an icon 531, 532 is displayed to represent a leaf class 531 in step 524 or a subclass 532 in step 525. Attributes are displayed for the selected class in area 509 and control is returned back to the user in step 528. Classes are closed by double clicking on an open folder icon 190, this displays a closed folder icon 529 and collapses all subclasses of the selected class. Leaf classes do not have any subclasses and are displayed as document icons 531. Leaf classes 531 cannot be opened or closed.

A class can be reparented to a new subclass as described in the flow chart of **Figure 92** and in connection with **Figures 93-94**. The user selects the subtree to be moved in step 534 which is highlighted 544 in screen area 501. In step 535, the user holds the mouse button down 117 and the control key on the keyboard 122

and drags the class in area 501 onto the class that is to become the new parent of the selected class being dragged 544. As the user is dragging the selected class, the class being dragged over is highlighted and the mouse cursor is changed to a no drop icon in step 538 if the class is a sibling of the selected class being dragged in step 535. If the class being dragged over is not a sibling of the selected class 544 being dragged, the cursor is changed to a drop icon in step 539. When the user drops the selected class being dragged in step 540 on a legal drop class, the knowledge base 123 is updated to represent the new class structure in step 541. The class tree 501 is also updated to represent the new class tree 542 and 545. Control is then returned to the user 528.

A class can be rearranged within a subclass with sibling classes as described in the flow chart of **Figure 95** and in connection with **Figures 96-97**. The user selects the subtree 545 to be rearranged in step 547 from screen area 501. The user holds the mouse button down 117 and drags the class in area 501 onto the class that is to become the new location of the selected class being dragged in step 547. As the user is dragging the selected class, the class being dragged over is highlighted and the mouse cursor is changed to a no drop icon in step 551 if the class is not a sibling of the selected class being dragged in step 547. If the class being dragged over is a sibling of the selected class being dragged in step 547 the cursor is changed to a drop icon in step 552. When the user drops the selected class being dragged in step 553 on a legal drop class, the knowledge base 123 is updated to represent the new class structure in step 554. The class tree 501 is also updated to represent the new class tree 557 in step 555. Control is then returned to the user in step 528.

New classes are added using the add button 505 as described in flow chart **Figure 98** and in connection with **Figure 99 - 100.** The user selects a class in the class tree area 501 that will be used as the parent of the class to be added. The user selects the add button 505 and the add class dialog 564 appears in step 560. The new class title is entered into the dialog box in step 560. In this example "custom hardware" has been entered in text entry field 565. The user then selects either the OK button 566

or the cancel button 567. If the OK button 566 is selected in step 561 the new class is added to the knowledge base. The screen 501 is updated to show the new class tree 568 in step 562, as shown in **Figure 100**. The new class is a leaf class and is represented as a document icon 531. If the parent class was a leaf class the parent class icon will be changed to a open folder icon 530. The add class dialog 564 is closed in step 563 and control is returned to the user in step 528. If the cancel button is selected the add class dialog box 564 is closed in step 563 and control is returned to the user in step 528.

An attribute can be rearranged as described in the flow chart of **Figure 101** and in connection with **Figures 102** and **103**. In this example "finish" 579 will be rearranged under "head recess" 580. The user selects the attribute 579 to be rearranged in step 570 from screen area 509. The user holds the mouse button down 117 and drags attribute 579 in the attribute area 509 onto the attribute 580 that is to become the new location of the selected attribute being dragged in step 576. As the user is dragging the selected attribute in step 570, the attribute being dragged over is highlighted in step 572 and the mouse cursor is changed to a no drop icon in step 574 if the class is an inherited attribute. See step 573. If the attribute being dragged over is not an inherited attribute the cursor is changed to a drop icon 575. When the user drops the selected attribute being dragged in step 576 on a legal drop attribute, the knowledge base 123 is updated to represent the new attribute structure in step 577. The attribute area 509 is also updated to represent the new attribute structure 579 in step 578 as shown in **Figure 103**. Control is then returned to the user in step 528.

A new enumerated attribute can be added as described in the flow chart of **Figure 104** as shown in **Figure 105**. In this example a new enumerated attribute titled "material" is added. The user selects the add button 513 from screen area 509. The add attribute dialog 588 is displayed in step 582. The user selects the type of attribute to add, in this example enumerated 589 is selected in step 583. In step 584, the user then enters an attribute title to represent the enumerated attribute, in this example the user entered "material" 590. The user can then select either the OK button or the cancel button in step 585.

If OK is selected, the knowledge base is updated and the attribute list in area 509 is updated to include the added attribute in step 586 and the add attribute dialog is closed in step 587. Control is then returned to the user in step 528. If the cancel button is selected, the add attribute dialog is closed in step 587 and control is returned to the user in step 528.

A new numeric attribute can be added as described in the flow chart of **Figure 106** as shown in **Figures 107 - 108**. In this example a new numeric attribute titled "length" is added using unit family inches. The user selects the add button 513 from screen area 509. The add attribute dialog 588 is displayed in step 582. The user selects the type of attribute to add, in this example numeric 599 is selected in step 594. In step 584, the user then enters an attribute title 600 to represent the numeric attribute, in this example the user entered "length" 600. The user can then select either OK or cancel in step 585. If OK is selected, the unit family dialog 1600 is displayed in step 595. The unit family dialog 1600 contains a list of all available units 1601 for the entire knowledge base 123. If the OK button 1602 is selected from this dialog box 1600, a new numeric attribute of unit type length is knowledge base and the attribute list is updated in step 598. Control is then returned to the user in step 528. If the cancel button 1603 is selected, the add attribute dialog 588 is closed in step 587 and control is returned to the user in step 528.

A new Boolean attribute can be added as described in the flow chart of **Figure 109** and as shown in **Figure 110**. In this example a new Boolean attribute titled "purchased" is added. The user selects the add button 513 from screen area 509. The add attribute dialog 588 is displayed in step 582. The user selects the type of attribute to add, in this example Boolean is selected 1605 and 1607. The user then enters an attribute title to represent the Boolean attribute, in this example the user entered Purchased 584 and 1606. The user can then select either OK or cancel 585. If OK is selected, the knowledge base is updated and the attribute list is updated to include the added attribute 586 and 509 and the add attribute dialog is closed 587. Control is then returned to the user 528. If the cancel button is selected, the add attribute dialog is closed 587 and control is returned

to the user 528.

A new string attribute can be added as described in the flow chart in **Figure 111** and screen shot **Figure 112**. In this example a new string attribute titled Manufacturer is added. The user selects the add button 513 from screen area 509. The add attribute dialog is displayed 582 and 588. The user selects the type of attribute to add, in this example string is selected 1609 and 1611. The user then enters a attribute title to represent the string attribute, in this example the user entered Manufacturer 584 and 1610. The user can then select either OK or cancel 585. If OK is selected, the knowledge base is updated and the attribute list is updated to include the added attribute 586 and 509 and the add attribute dialog is closed 587. Control is then returned to the user 528. If the cancel button is selected, the add attribute dialog is closed 587 and control is returned to the user 528.

Enumerators for enumerated type attributes can be added and inserted as described in the flow chart **Figure 113** and as shown in **Figure 114** and **Figure 115**. When an enumerated attribute is active in screen area 509 the edit button 515 is activated. When the edit button 515 is selected, the edit enumerator dialog box 1620 is displayed with a list of enumerators for the selected enumerated attribute 1613. The user can either select the add button 1621 or the insert button 1622. If the add button 1621 is selected in step 1615, a blank line is added after the active enumerator in the dialog box 1620 and the knowledge base 123 is updated. If the insert button 1622 is selected in step 1616, a blank line is added before the active enumerator in the dialog box 1620 and the knowledge base 123 is updated. The enumerator title is typed into the blank line in dialog box 1620 in step 1617, in this example "aluminum" is entered and the knowledge base is updated in step 1617A. In the example on **Figure 115**, "steel" has been added and the insert button 1622 was selected to add a blank line above the enumerator "steel." When the user has completed adding/inserting enumerators, the close button 1623 is selected in step 1619, and the edit enumerators dialog box 1620 is closed. The control is then returned to the user in step 528.

Enumerators for enumerated type attributes can be deleted

as described in the flow chart of **Figure 116** and as shown in **Figure 117**. When an enumerated attribute is active in screen area 509 the edit button 515 is activated. When the edit button 515 is selected, the edit enumerator dialog box 1620 is displayed with a list of enumerators 1624 for the selected enumerated attribute 1613. The user selects an enumerator in step 1626 then selects the delete button 1629 in step 1627. In step 1627A, a confirmation dialog box 1630 displays allowing the user to select either the "yes" button 1631 in step 1627C or the "no" button 1632 in step 1627D. If "yes" is selected, the enumerator is removed from the edit enumerator list 1624 and the knowledge base is updated in step 1627C and the confirmation dialog is closed in step 1627D. If the user selects "no" in step 1627D, the confirmation dialog is closed. When the user has completed deleting enumerators, the close button 1623 is selected in step 1619 and the edit enumerators dialog 1620 is closed. The control is then returned to the user in step 528.

Figure 118 describes the functions that can be performed from the numeric table editor dialog box 1550 in **Figure 119**. This dialog box 1550 allows the user to build tables of numeric values for a numeric attribute. The numeric table editor dialog 1550 is invoked in step 1500 from the schema editor 500 after selecting a numeric attribute such as 1552. The edit button 515 invokes the table editor dialog 1551.

In step 1501, calls are made through the API 143 to display existing table data. If no table data exists, a table 1554 with 1 row and 1 column is constructed and displayed as shown in **Figure 119**.

Tables 1554 consist of cells which have numeric values and labels associated with them. A label is distinct from a value, and is used as a textual description or representation of the underlying values. Table cells 1554 must contain ascending numeric values. Labels may be in any collating order.

In **Figure 118**, the user adds values to a table in step 1504 by executing the procedure described in **Figure 120**. The user may optionally label a table manually in **Figure 120**, step 1509, or use the auto-label feature in step 1510 by selecting item 1559 in **Figure 121**. The auto label button 1559 invokes the automatic values dialog box 1560 in step 1510. In step 1511, the user fills

in values for items 1561, 1562, and 1563 and selects the OK button 1564. In step 1513, the values for the cells are calculated and set in the table. In step 1514, the automatic values dialog 1560 is closed, and control is returned to the user in step 1515.

In **Figure 120**, if the user chose to label the table cells manually in step 1509, the user selects item 1565 and enters a value, accepted by selected check box 1566A. Any cells the user had selected in item 1566 are filled with the value in step 1517. Control is returned to the user in step 1515.

In **Figure 118**, the user performs step 1503 to add labels to the table. The process for adding labels to the table is described in **Figure 122**. In step 1519, the user may select auto label or manual labeling. If auto label item 1567 in **Figure 123** is selected, the automatic labeling dialog 1568 is invoked in step 1520. For each column in the table 1569, the user may type in a label. In step 1522 the user may select the OK button 1570 or the cancel button 1571. If the user selects the OK button 1570, step 1523 sets the cell labels to the current cell value concatenated with the label values from the automatic labeling dialog 1568. Dialog 1568 is dismissed, and control is returned to the user in step 1524.

Tables are structured as rows and columns. The user may wish to change the number of columns and rows in a table by executing step 1502. Rows and columns are entered by using the edit boxes 1555 or 1556 and the check mark button 1557. To not accept a value, the "x" button 1558 may be selected. The procedure for changing the number of columns and rows is described in **Figure 124**.

In **Figure 124**, the user may select rows in step 1536 by selecting item 1556 in the table editor dialog 1550. The user enters the number of rows in item 1556 and selects item 1557 to accept the item. In step 1537, the number of rows kept internally is adjusted to the number of rows entered in step 1536. The number of rows is checked in step 1538. If the number of rows is greater than the number of rows previously in the table, the new rows are added with the default value of "0" and no labels in step 1540 and control is returned to the user in step 1535. If no new rows are needed, "0" or more rows are

deleted from the table along with their labels in step 1539 before returning control to the user in step 1535.

In **Figure 124**, the user may have selected columns to change in step 1530 by selecting item 1555 and entering a numeric value and selecting item 1557 to accept the item. This new number is set internally in step 1531. If new columns need to be added as determined in step 1532, step 1534 adds new columns to the table with the default value of "0" and no labels. If no new columns are needed as determined in step 1532, "0" or more columns are removed from the table along with their labels.

Back in **Figure 118**, the user closes the table editor in step 1506 by selecting the OK button 1572 in **Figure 119** or by canceling changes by selecting cancel button 1573 in **Figure 119**. The table editor dialog box 1550 is dismissed in step 1507 and control is returned to the user in step 528.

In **Figure 127**, the process for deleting an attribute from schema editor 500 is shown. In **Figure 128**, the user selects the mechanical class 2206. Note that all the cells in area 509 are dimmed, and the delete button 519 is not active. The schema editor 500 only allows the attributes defined by mechnical class 2206 to be edited, and there are no locally defined attributes.

In **Figure 129**, the user selects test hardware item 2207. Attributes are defined at this class and in area 509, the local attributes in items 2208, 2209, and 2210 are not dimmed and are available for editing. The user selects item 2214 in **Figure 130** and it is highlighted.

In step 2200 of **Figure 127**, the user selects the delete button 519 in **Figure 130**.

In step 2201 of **Figure 127**, the dialog box 2211 in **Figure 130** is displayed to allow the user to verify the deletion of the attribute 2214. If the user selects button 2213 in **Figure 130**, the dialog box 2211 is dismissed in step 2204 and control is returned to step 528 from step 2205.

If the user selects button 2212 in **Figure 130**, the attribute is deleted from the knowledge base in step 2203 and item 2214 of **Figure 130** is deleted from the display area 509 **Figure 130**. Step 2204 is executed to dismiss dialog 2211, and control is returned to step 528 from step 2205.

H. Legacy and the Legacy Manager

The legacy manager 145 is a component of the dynamic class manager 134 that provides services for classifying, and parameterizing data. Legacy 133 is an application that provides a graphical interface and tools for classifying, parameterizing, moving, importing, and editing parts, a process also known as "legacizing". Legacy 133 communicates with the knowledge base client 131 using the API 143

Figure 132 shows a preferred process for performing the transformation of customer legacy parts data into a parts knowledge base in a form usable by the dynamic class manager 134, thereby providing access to users through a retriever 130.

In step 600, customer parts data sources, which may include data from material requirements planning systems, part master systems, bill of material systems, purchasing systems, engineering drawing systems, part catalogs, crib sheets, intelligent part numbering systems in files 601 are analyzed for possible complete or partial inclusion in the input files that serve as input to legacy 133. Parts data sources to be used in legacy processing are segregated into legacy input files 602. These original parts data sources may be in a variety of formats, including fixed length records, delimited records, COBOL file formats, or others which are converted in step 603 to importable legacy data files 604, which consist of text identified by part identifier with fields separated by a standard delimiter, usually an ASCII tab character.

In step 605 the legacy input files are analyzed to determine if data augmentation would be appropriate. For example, classification and parametric information about integrated circuit parts is available referenced by manufacturer and device number and may be used to augment or replace any other descriptions of these parts by use of genic 3000. Similarly, classification and parametric information is available from government and industry standards, or customer supplied engineering tables, which may then be automatically merged with other descriptions of these parts. The resulting optionally augmented part legacy data is stored in files 607.

Step 608 includes running the classify program to perform initial classification of the optionally augmented legacy data

607. In addition, if parts data is to be imported to classes based on patterns identified in the parts data, import maps are generated describing the relationship of the patterns to their associated classes. Finally in step 608, any required custom schema development is performed by a combination of manual means and use of the schemagen program. The result of step 608 is the creation of a preliminary knowledge base accessible by the dynamic class manager 134 and therefore legacy 133.

In step 610 the graphical user interface of legacy 133 is used by subject matter knowledgeable users who are assigned to perform further part classification and parameterization on parts within identified subtrees of the class hierarchy. By iterative application of legacy 133, the preliminary customer parts knowledge base 611 is produced. In step 612 a combination of random sampling, use of the ability of retriever 130 to query on parts at a non-leaf class in the schema to identify partially classified parts, querying on undefined attributes to identify incomplete parameterization, and sorting and inspection of parameter minimum, maximum, and standard values is used for quality inspections and correction of part classification and parameterization. The final result of the quality assurance activities performed in step 612 is a customer parts knowledge base ready for delivery to the customer. In step 613, this knowledge base is delivered to the customer by means of computer tape, disks, or other computer-readable means, with the delivered knowledge base 614 being further maintained and enhanced by the customer through retriever 130.

In the present invention, legacy 133 provides graphical user interface to the classify a part function 1101, parameterize a part function 1102 of the legacy manager 145, along with software programs for performing initial part classification 3001, a schema generation program 3002 for custom schema generation from data, and genic 3000, a data augmentation through analysis, lookup, classification and parameter generation for integrated circuit type parts based on manufacturer and device identifiers.

Legacy 133 includes the query formulation and part display and editing functions of retriever 130 as a means of querying, displaying and modifying the parameters of parts, including selecting those parts to be classified and parameterized, and as

a means of navigating the class hierarchy and associated attributes to select classes, attributes and enumerators for thesaurus editing.

Legacy 133 also provides a graphical user interface for the creation, modification and deletion of thesaurus entries stored as metadata associated with classes, numeric attributes, boolean attributes, enumerators of enumerated attributes, and units within unit families. Legacy 133 also includes a means for setting and modifying the class types collection, primary, and secondary which are used to control the classify a part function 1101 in its use of the class hierarchy.

It provides a means for selecting the source attribute 1266, a text attribute from which the text parameter to be analyzed by the legacy manager 145. It also provides a means for selecting the destination attribute 1267, the text parameter of is set to return to the user the text resulting from application of thesaurus entries when a part is classified or parameterized.

The user may specify a list of attributes to parameterize 1277 which is defined by the legacy manager as a superset of the parameters that may be set during parameterization. Legacy 133 provides a graphical user interface for adding and deleteing parameters from the list of attributes to parameterize 1277.

The user may also specify a virtual root 1269 which is defined by the legacy manager as the class from which legacize ancestor classes 1112. By this means, the user may effectively control which superclass thesaurus entries are applied to legacizing one or a group of parts.

Figure 170 shows how the user accesses knowledge bases for legacy processing by step 615, selecting Open from the drop down menu choice File 1201 in **Figure 171**.

The registry server 141 is queried for a list of knowledge bases and rights available to the user in step 616. The results are displayed to the user in step 617 as a selectable, scrolled list 1200 specifying the knowledge base name 1202, with rights to retrieve parts 1203, edit parts 1204, edit schema 1205, and make parts 1206 shown for each knowledge base known to the registry server 141. When a user selects a knowledge base, such as the example "fifi" 1202 shown with retriever 1203, edit parts 1204, and edit schema 1205 rights and with make part rights 1206

denied, the legacy button 1207 will be dimmed if the user does not have legacy rights to the selected knowledge base 1202. If the user has legacy rights to the selected knowledge base, the legacy 1207 button may be used in step 618 to continue with legacy 133, displaying the work area selection window 1212 shown in **Figure 172.**

In step 619 the user selects and locks the work area or cancels and does not select a work area. As shown in **Figures 172** and **173**, the class hierarchy is presented starting with the root of the knowledge base 1216 step 624. By manipulating the class hierarchy in step 625 as in retriever 130, selecting a class 1213 as the root of the work area, and using the work in area button 1214 to request the work area or cancel button 1215 in step 625. If the user cancelled the request as tested in step 626, the work area 1217 window is removed and the initial legacy window 1199 is displayed. If the user did not cancel the request as tested in step 626, a subtree lock is requested for the selected class in step 627. If the lock is not granted as tested by step 628, an error dialog informs the user that another user is working in the requested subtree in step 631. If the lock is granted, a retriever 130 with additional legacy functions is invoked in step 629 with the class hierarchy rooted at the class selected by the user 1216. Control is returned to the user in step 630.

Legacy 133, with part specification window 1224 shown in **Figure 174** includes the functions of retriever 130 as discussed in A. Retriever the following differences. The available parts in a the subtree of a selected class 1218 are shown, with the value independent of any query selectors set as search criteria. An update count button 1220 is provided to allow the user to explicitly request a query to be performed with the matching part count updated as parts found. This function is provided separately in legacy because the processing of legacy knowledge bases begins with parts either unclassified or roughly classified to high levels in the parts hierarchy, resulting in possible performance penalties for some queries. By allowing the user control over when a query other than the query that returns the part count for a subclass is performed, significant additional effeciencies may be achiever, especially during the early part of a legacy processing project.

The class hierarchy 1223 displayed by legacy includes additional icons 1222 that show classes with locks that may have been applied by other users. These locks may be refreshed at the user's request, providing feedback concerning the work areas of other legacy and schema editor users and allowing class access conflicts to be more easily resolved.

Legacy 133 also provides a user interface for class thesaurus editing as shown in **Figure 175**. The process of editing a class thesaurus is shown in the flowchart in **Figure 176**. In step 631, the user navigates to and selects a class 1225 and chooses thesaurus entry editing 1226 from a drop down menu available by use of the right mouse button. In step 632, the thesaurus list is obtained from metadata for the class 1225 through the dynamic class manager 134. In step 633 the user edits the thesaurus using the thesaurus editor 1227 shown in **Figure 177** before returning to the retrieve parts window 1228. In the example shown in **Figure 177**, the thesaurus currently has one entry 1229, which contains a regular expression suitable for matching many of the common text forms for describing 1/4 inch 20 threads per inch pitch machine bolts. In the editing example, the user adds a thesaurus entry for the same size machine bolts where "1/4 inch" is described as ".25 -" or some variant thereof.

Thesaurus editing consists of modifying the list of text strings in a thesaurus using the controls provided by the thesaurus editor 1227 as shown in the flow chart in **Figure 178**.

The user selects one of seven thesaurus editor actions in step 635. If the test at step 636 determines that the user selected the cancel button 1230, control is returned to the invoking window without updating the thesaurus being edited. If the OK button is chosen, the thesaurus for the schema object is replaced with the text in the thesaurus editor in step 642. The add button 1233 is used in step 639 to open a blank line 1237 below the currently selected thesaurus entry as shown in **Figure 179**. The copy button is used to store the contents of the currently selected thesaurus entry 1229 in step 637 so that it may be used to replace a selected thesaurus entry 1237 as shown in **Figure 179**. In the example shown, the user replaces "1/4" 1238 in the new thesaurus entry 1237 with "\.250*" in **Figure 181**. In this way, the user can easily reuse thesaurus entries to create

patterns that match similar forms of text that may be found in part descriptions for parts of a class.

In **Figure 182** a blank thesaurus entry 1240 has been created by using the insert button 1234 and step 640 with thesaurus entry 1237 selected. Deleting thesaurus entry 1240 using the delete button 1235 and step 641 would result in the thesaurus entries shown in **Figure 181**.

Figure 183 shows a thesaurus entry of a type that is likely to be found at a nonleaf class in the class hierarchy. It matches the portion of a standard form of text description of a fractional sized machine bolt, transforming the portion of a string such as "0.25-20 X 2.5 L, CAP HD, STLNPHEX SKT" into "0.25-30 length={0.75 inch}L,CAP HD,STLNPHEX SKT", from which it is significantly easier and more reliable to automatically extract the length parameter with the correct unit of inches. **Figure 183** also shows that the thesaurus editor 1227 can be invoked by selecting the the thesaurus drop down menu for a class in the class hierarchy accessible from either the part specification window 1224 or the part editing window 1243.

The flowchart in **Figure 184** shows the process of editing thesauruses for the enumerators of an enumerated attribute. In the example shown in **Figure 185**, the enumerated attribute Finish 1244 is selected and the drop down menu including the thesaurus entry choice is selected as step 648. The result is shown in **Figure 186**. The enumerator list 1247 is obtained for the selected attribute 1244 and displayed in step 649. Selecting an enumerator such as the example "Cadmium Plate" 1248 invokes the thesaurus editor 1227 for that enumerator's thesaurus, which functions as described in the flowchart in **Figure 178.** The thesaurus editor for enumerator thesauruses may also be invoked from the column heading of the edit parts window 1243 as shown in the example in **Figure 187** where a thesaurus entry 1251 for the enumerator "Black Oxide" 1250 for attribute Finish 1244 is being displayed and edited.

The flowchart in **Figure 203** shows the process of editing a thesaurus entry for a numeric, text or boolean attribute. In the example shown in **Figure 189** the thesaurus 1252 for a numeric attribute, Length, 1253 is edited. The user selects the attribute from either the part specification window 1224 or the part

editing window 1243 as shown in **Figure 190** by using the drop down menu 1246. The thesaurus for the selected attribute is obtained in step 656 and edited by the thesaurus editor 1227 as shown in **Figure 191**, with control return to the user in step 658. The example thesaurus entry is a regular expression that will match some standard forms of machine bolt length descriptions as transformed by class thesaurus entry 1241 in **Figure 183**.

Figure 192 shows thesaurus entry editing for units within a unit family. In the associated flow chart in **Figure 193**, the user selects the unit thesaurus editing button 1254 from the legacy tools toolbar 1255 in step 659. The list of all unit families is obtained and presented to the user in a drop down list 1256 in step 660. In step 661, the user may return from the unit thesaurus editor by choosing the OK button 1258 or cancel button 1259 through step 666. If instead the user selects a unit family from the drop down list 1256, step 662 obtains a list of derived units for the unit family 1260. The user selects a derived unit 1260 in step 663 and step 664 obtains the thesaurus for the unit 1261, which is then edited with the thesaurus editor 1227.

Figure 96 shows the processing of queried parts by the legacy functions classify a part 1101 and parameterize a part 1102. **Figure 195** is a flowchart describing the process by which parts selected 1262 from the part editing window 1243. In step 667 the user selects part 1262 from the attribute display and chooses the legacy processing window from the tools menu 1264. In step 668 the legacy processing window displays the class path from the root of the users workspace to the class of the current query. Also displayed are drop down lists of available source attributes 1266 and destination attributes 1267. A list of available target attributes for parameterizing are displayed if the parameter setup button 1268 is chosen. The user may also select a virtual root 1269 for controlling the application of ancestor class thesaurus entries during classify a part 1101.

If the user chooses the legacize button 1270, the selected parts are both classified 1101 and parameterized 1102, with resulting part parameter values displayed in the part display window 1262. **Figure 196** shows the result of legacizing the selected parts 1279. In the first line of the parts display 1281

after legacizing, the finish parameter is set to "Cadmium Plate" due to matching the thesaurus entry 1249 "CAD[MIUM PLATE]* from **Figure 186.** The length 1282 is set to .5625 inches due to matching a combination of the class thesaurus entry at the class Fractional, the numeric attribute thesaurus entry for length, and the unit thesaurus entry for the unit "inches".

If the user chooses the classify button 1271, the selected parts are classified 1101. The results of classifying a part may be inspected by using the part information button 1273.

If the user chooses the parameter setup button 1268, the process described in the flow chart in **Figure 197** displays attributes for the class of the current query in step 678. In response to choosing the insert button 1274, step 680 inserts the selected available attribute 1276 into the list of attributes to parameterize 1277 above the current selection. In response to choosing the add button 1275, step 681 adds the selected available attribute 1276 into the list of attributes to parameterize 1277 below the current selection. In response to choosing the remove button 1278, step 682 removes the selected attribute to parameterize 1277. The result of editing the attributes to parameterize 1277 are shown in **Figure 198.**

The legacy manager 145 is a component of the dynamic class manager 134 which automatically subclassifies and parameterizes an instance based on a combination of text data in a source attribute 1266 and thesaurus entries that may be available as metaparameters to classes, text attributes, enumerators or enumerated attributes, boolean attributes, numeric attributes, and units. Classification by the legacy manager 145 is accomplished by the classify a part function 1101, a non-parsing method employing matching of thesaurus entries interpreted as regular expressions against source attribute 1266. Each successful match increases the score for the class at which the matching class, attribute, or enumerator thesaurus entry was found. The thesaurus entry matches are performed recursively down the class hierarchy, beginning with the class at which the part instance is currently defined, with scores being compared in the style of a single-elimination tournament as the recursive calls return. If there is a clear winner among sibling classes, the winner is passed up in the recursive call return to compete at

the next level. If a class achieves a score equal to the current winner within a sibling group, the winning score is stored and the current winner is marked as tainted and may not be declared the winner for the sibling class group. However, a sibling class that achieves a superior matching score to the tainted winner score will be declared the winner of the sibling group. If there is no winner within a sibling group, the superclass of that group is declared the winner and competes with its siblings in the next round of competition. When the recursive descent of the class hierarchy is completed, if a winner has been chosen among the classes in the subtree, the part instance being classified has its owner set to the winner class. The part instance is then analyzed again, first by having the thesaurus entries between its owner and either the root of the class hierarchy, or a virtual root class supplied when invoking the classify part function 1101, applied to the text parameter defined by source attribute 1266, which may result in eliding portions of the text. This modified text is used to set the text parameter defined by destination attribute 1267, providing feedback to the user concerning the combination of thesaurus entry matches that were used to classify the part instance.

Turning to **Figure 133**, the automatic part classification function of the legacy manager begins by insuring that the source and destination attributes chosen by the user are text attributes and are either local to the part instance's owner class, or are inherited at that class in step 1104. If an illegal attribute is detected in step 1105, the part instance is returned with its classification unchanged, else initialization is performed by creating a local copy of the source attribute text parameter and initializing the legacy attribute in step 1106, a local copy of which contains the score of the class at each node of the class tree as the classification tournament progresses. The instance is then classified in step 1107, following the method outlined above. If the resulting working string is determined to be of zero length in step 1108 due to the application of thesaurus entries throughout the subtree in step 1107, the destination attribute's parameter is set to undefined for this part instance in step 1109, else it is set to the value of the working string in step 1110. The classified instance is returned to the caller

in step 1111.

In **Figure 134**, the method for classifying a part referred to in step 1107 is displayed. In step 1112, ancestors of the current class are legacized, with thesaurus entries being applied in order from the root class or a virtual root class supplied by the user. Then the subtree of the owner class of the instance is recursively descended to legacize the instance for the purpose of finding the class that provides the best overall match to the source attribute 1266 in step 1113. If the winner class differs from the current owner class for the part as determined by step 1114, the owner for the class is set to the winner class in step 1115, after which the classified part instance is returned in step 1116.

Legacizing ancestor classes is shown in **Figure 135**, with step 1113 ascending the class hierarchy from the owner class of the part to and including the root or virtual root class, creating a list which is ordered from root to owner class of the part. The first class in this list, which is the root or virtual root class, is obtained in step 1114. If step 1114 successfully obtained a class to process as determined by step 1115, the thesaurus entries for that class are processed in step 1116. This processing may result in modifications to the working string. The next class in the list is obtained in step 1117, with control then returned to step 1115, providing for a loop that processes each class in the list. When the loop terminates by encountering the end of the class list, the legacy attribute with updated scores and the modified working string are returned to the caller.

The method for legacizing an instance to determine the best matching class in the subtree is shown in **Figure 136**. First the working string is checked for zero length in step 1117, with step 1118 setting the winner class to the current class and returning if the working string does not contain any characters which could influence the further choice of a matching class. If the working string has one or more characters, the thesaurus entries for the current class are processed in step 1119. If none of the thesaurus entries applied in step 1119 matched the working string, the class type is tested in step 1122. If the class type is marked as primary and the owner class of the instance is not

the same as the current class as tested in step 1123, processing continues through step 1125 and the current class is returned as the winner of this subtree. This is done to insure that subtrees for primary classes are not descended unless the primary class has at least one matching thesaurus entry, preventing unnecessary processing. If the class type is marked as collection, thesaurus entries only must be matched if they are found in step 1124 - a collection class with zero thesaurus entries of the type that are intended to match and elide parts of the working string will always have its subclasses explored for a better match. If not, processing continues through step 1125 with the current class being declared with winner for the subtree. Classes marked as secondary are descended whether or not any thesaurus entries matched the working string.

In the cases where the rules as described above for continue the descent of the subtree are met, thesaurus entries for local attributes of the current class are processed in step 1126. In step 1127, the winner for the tournament over the subtree rooted at the current class is provisionally declared to be the current class. Processing then continues through step 1128 to **Figure 137**, where the subtree will be explored for a better match. This is accomplished by recursive descent of the subtree, which begins with getting the list of subclasses for the current class in step 1129. As long as this list is not fully processed, as tested in step 1130, a loop is executed in which the next class in the list is obtained in step 1131 and the legacize instance function 1113 is recursively called. This function always returns with a winner class set, which may be the same as the current class in the case that a superior match was not found in the subtree. The score for the winner class so returned is compared to the current winner in step 1114. The count of primary class matches is weighted most heavily, followed by secondary and collection class matches, non-numeric attribute matches, numeric attribute matches, and finally, as a tie breaker, the class with the shorter length of working string after processing is preferred. If the returned winner class has a lower score than the current winner, it is rejected and the loop continues with step 1130. If the returned winner score is identical to the current winner, the current winner is marked as tainted in step 1115 and will not be

allowed to be declared the winner for this subtree. This is to prevent an equal match across all subclasses from favoring the first subclass processed. If the returned winner has a higher score than the current winner, it is stored as the current winner in step 1116 and further competition within this subtree is with this new winner class.

When the list of subclasses has been processed as indicated by the test in step 1130 failing, processing continues through step 1125. In **Figure 138**, the tainted winner flag is tested in step 1186, with the current winner being rejected if the flag is set, resulting in the current class being declared as the winner of its subtree in step 1187. If any of the processing resulted in a new winner being declared as tested in step 1188, the current working string is replaced with the working string returned from the competition in the subtree in step 119. The current score is updated with the new winner score in step 1190, with this score being used to compete with classes in the next level of the tournament. Regardless of whether a new winner was declared, the final winner class for the subtree is returned to the caller in step 1121.

In **Figure 139**, the processing of attributes for the purpose of classification is shown starting with obtaining the list of local attributes for the class in step 1131. These attributes are all processed in a loop controlled by the test in step 1132, with step 1133 getting the next attribute in the list for analysis. The details of the matching of an attribute's thesaurus entries to the working string are controlled be the test for attribute type in step 1134.

Enumerated attributes do not themselves carry thesaurus entries. The terms associated with enumerated attributes that are likely to be found in a part description are the possible value of the attribute, the enumerators. Each enumerator may have a list of thesaurus entries, any one of which may match a part of the working string. In order to test each enumerator, a list of enumerators is created in step 1135. As long as the test for another enumerator in step 1136 succeeds, the thesaurus, consisting of a list of thesaurus entries, is obtained in step 1137, and processed against the working string in step 1138. If an entry in the thesaurus matched, the non-numeric attribute

score is incremented for the class in step 1140, improving its degree of match against the working string. If no thesaurus entry matched, the loop continues, processing each enumerator in order.

If the attribute is a text string or boolean type, it may have a thesaurus containing thesaurus entries describing both how to transform the working string locally before further processing, and what patterns constitute a match for the text string or boolean attribute if matched in the working string. This thesaurus is obtained in step 1141 and processed in step 1138, using the general function for processing any thesaurus for any schema element. If the thesaurus entry matched, the non-numeric attribute score is incremented for the class in step 1140.

Numeric attributes must be evaluated both in terms of the numeric thesaurus itself, and any associated unit thesaurus. The numeric thesaurus entries attempt to match a combination of digits and other numeric symbols, along with patterns that would indicate an appropriate context for finding the numeric information in a text string. The unit thesauruses for the base and derived units of the unit family for the attribute may contain patterns that discriminate among the different convertible units that might be found in the working string, such as "in" for inch or "ft" for feet, allowing the legacy manager to correctly interpret and convert such information. The thesaurus for the numeric attribute is obtained in step 1141 and processed in step 1138. If no thesaurus entry matched as tested in step 1139, no further processing is done, else the list of units for the unit family for the numeric attribute are obtained in step 1142. A loop controlled by the test for another unit in the list in step 1143 gets the thesaurus for the next unit in step 1144, processes that thesaurus against the working string in step 1138, and tests for a match in step 1139. A match indicates that a number and unit in correct combination have been found in the working string, increasing the numeric attribute score for the class in step 1145.

Figure 140 shows the generalized mechanism for processing a thesaurus against the working string for all types of schema objects that may have thesauruses defined: classes, numeric attributes, enumerators, boolean attributes, and text attributes.

It is invoked with a flag to indicated whether a successful match of a thesaurus entry should result in modification of the working string. Processing beings by getting a list of all the strings, or thesaurus entries, that make up the thesaurus provided by the caller in step 1146. These entries are processed in a loop that is controlled both by the test for another thesaurus entry in the list in step 1147, and the test for a successful pattern match in step 1157. In each iteration through the loop, the next thesaurus entry is processed against the working string.

There are two general types of thesaurus entries used to match the text in the working string. The first, called a modifying or editing thesaurus entry, begins with either a "v/" or a "g/". These thesaurus entries behave essentially like editing commands in the UNIX vi editor. For example, a thesaurus entry "g/X .*([0-9.][0-9.\/][0-9.\/]*)/s// length={\1 inch}" would only act on a string that matched regular expression pattern between the first pair of slashes, and would substitute for that string the text "length=", followed by the text in the working string matching the captured portion of the pattern (between the open and closed parentheses), followed by the word "inch". For example, this thesaurus entry applied to the working string "1/4-20 X 1.25" would produce the resulting working string "1/4-20 length={1.25 inch}". Providing this type of thesaurus entry, employing full regular expressions with the ability to capture and reuse portions of the working string, allows for the reliable evaluation of standard form text fragments common to a particular class or attribute without requiring that all of the text conform to a correct or canonical form, thereby allowing legacy to successfully exploit a wide variety of data of the sort commonly found in legacy parts data sources.

The "v/" form of this type of thesaurus entry works identically to the "g/" form, with the exception that the filter, or first pattern, is considered to match if the regular expression it contains is not matched in the working string. This allows the selective modification of working strings that are discovered to be missing data that, if provided, would allow for simpler processing by later thesaurus entries.

The processing for editing style thesaurus entries begins by extracting the filter between the first pair of slashes in

step 1148. The pattern to be matched is then extracted in step 1149. If the pattern is missing, it is defaulted to be identical to the filter, as in the example above. A flag is set to avoid later modification of the working string other than the transformation defined by the thesaurus entry itself in step 1151. If the filter matches the working string, keeping in mind the opposite sense this has for "v/" thesaurus entries, as tested in step 1152, then the modify flag is set and the pattern is tested against the working string in step 1157. If it matches, and the caller requested that the working string be modified and the modify flag is set, as tested in step 1158, the text in the working string matched by the pattern is replaced with the replacement text, with appropriate expansion of captured text, in step 1159. If a thesaurus entry matched, a boolean value of true is returned to the caller in step 1160.

Non-editing, or simple, thesaurus entries are intended to match and optionally result in eliding matched text from the working string. They are distinguished by not starting with "v/" or "g/". The pattern is set to the thesaurus entry in step 1154, and the flag is set to modify the working string in step 1155. The replacement text is set to a single character "!" in step 1156 that can be detected in later thesaurus entries, providing a simple means for determining either by inspection of the destination attribute 1267 or by a thesaurus entry matching "!". In this way, the effect of matching a thesaurus entry can be made conditional upon the successful application of an earlier thesaurus entry. From this point, processing continues as for the editing thesaurus entries, with a test for the pattern being matched in the working string in step 1157, followed by optional replacement of the matched text by the character "!", defined as the current replacement text in step 1159 as controlled by the test in step 1158.

If the entire list of thesaurus entries is processed without a match as indicated by the test in step 1147 failing, a boolean false is returned to indicate that no matches occurred.

Processing the thesaurus for a class is shown in **Figure 141**, where step 1116 first determines if the class is a collection class or not. Collection classes may contain a thesaurus, which may be used to avoid descending a subtree in order to tune

performance or improve reliability by limiting the scope of matches. However, while editing thesaurus entries may always result in the modification of the working string, simple thesaurus entries are not so used for a collection class, so a flag is set to avoid processing of the working string in step 1164. Conversely, for primary or secondary classes, simple thesaurus entry matches should always result in eliding the matched pattern from the working string, and the flag to process the string is set accordingly in step 1163. In either case, the function to process the thesaurus for a schema object is called for the class in step 1165, with the result returned in step 1166.

In addition to automatically subclassifying by matching both class and attribute thesaurus entries within a subtree to the text parameter defined by the source attribute 1266, one or more specified attributes may be automatically parameterized using thesaurus entries.

Parameterize a part 102 is shown in **Figure 142**. Like classify a part 1101, it begins by checking that source and destination attributes are local or inheritable to the owner class of the instance being parameterized in step 1104. If an illegal attribute is detected in step 1105, the instance is simply returned in step 1169. The text parameter for the source attribute is copied to the working string and the legacy attribute is initialized in step 1106. In order to preserve the effect of any thesaurus entries in classes between the owner class of the instance and the root, legacize ancestor classes is performed in step 1112, and the thesaurus entries for the owner class are also processed in step 1119. Non-numeric parameters are legacized first in step 1167, followed by numeric parameters in step 1168. This order allows simpler definitions of collections of thesaurus entries with less conflict between the numeric thesaurus entries and numeric data that is often found within enumerators or other non-numeric attributes. For example, the ceramic capacitor dielectric "X7R" contains a digit that might be inadvertently elided by applying a simple numeric thesaurus entry, but would be protected by allowing the non-numeric attributes to be processed first. After all parameters have been legacized, the parameterized instance is returned in step 1169.

To legacize non-numeric parameters, step 1167, the process in **Figure 143** is followed. In step 1170, the list of all inherited and local non-numeric attributes for the owner class of the part instance is obtained. The list may be further limited to those target attributes selected by the user using **Figure 194**, dialog 1263. Iteration through this list is controlled by the test for another attribute in the list by step 1132 in **Figure 145**, and successful thesaurus entry matches that may occur in step 1139. In this loop, the next attribute from the list is obtained in step 1133 and its type is determined in step 1134.

If the attribute type is enumerated, the same procedure of evaluating thesaurus entries for each enumerator is performed identically to that in **Figure 139**, with a list of enumerators being created in step 1135, then looped through until list is exhausted as tested by step 1136. For each enumerator, a thesaurus is obtained in step 1137 and processed in step 1138. If a thesaurus entry matches as tested by step 1139, the enumerated parameter for the attribute may be set to the enumerator for which the thesaurus entry matched. However, the parameter will only be set if it is currently undefined. Parameters which currently are set to a value are presumed to have been set by a more trusted prior process, either in the form of directly imported data, or a value entered by a human through either retriever or the part editing capability of the legacy interface.

A similar method is used for text string and boolean attributes as detected by the test in step 1134. The thesaurus for the attribute is obtained in step 1141, and the thesaurus is processed against the working string in step 1138. If a thesaurus entry matched and the parameter is currently undefined, it is set as appropriate, either to true if boolean, or to the matched result if a string attribute.

Legacizing numeric parameters, step 1168, is shown in **Figure 145**, starting with assembling a list of inherited and local attributes in the order of their definition in the schema in step 1175. The list may be further limited to those target attributes selected by the user using **Figure 194**, dialog 1263. While there remain attributes to be processed, the next attribute is obtained in step 1133, its thesaurus is extracted from its metaparameters

in step 1141, and processed in step 1138. If a thesaurus entry matched, the list of units for the unit family for the attribute is created in step 1142. Each unit is processed in a loop controlled by test 1143, with the unit obtained in step 1176 and the unit thesaurus obtained in step 1144 and processed in step 1138. A matching unit thesaurus entry as indicated by the test in step 1139 results in setting a the associated numeric parameter in step 1177. As in the case of non-numeric attributes, a currently set parameter value will not be overwritten by this step. When the list of numeric attributes is exhausted, the instance is returned in step 1178.

Legacy 133 also includes a classify program 3001 and a schema generator 3002.

The classify program as shown in **Figure 200** is used to match formal object names with human entered textual descriptions that contain abbreviations and spelling errors. The purpose of the classify program is to use the knowledge accumulated in the names of the schema objects in 2154 (class names, attribute names, unit families, etc.) to suggest locations that one might place a new part, or where one might go to look for a set of parts with a given description obtained in 2153. The classify program generates output which is a set of potential locations in the schema that the part description may be classified in 2157. The set of potential locations is reviewed by a human in 2158. The selected classification is then placed in an import map in 2159.

The classify program operates by using two word matching techniques. The first matching technique is referred to as the "Bickel Algorithm", and the other matching technique is referred to as the "Soundex" Algorithm.. These algorithms use different approaches to locate candidate word matches with the target word.

In the Bickel Algorithm, a mask describes the characters which are common to the target word, and the candidate word is scored based upon the frequency of the use of each character. The higher the aggregate score, the better the match. The Bickel Algorithm is well known to those skilled in the art.

In the Soundex Algorithm, a mask which describes the sounds of the characters used in both the target and candidate word is checked for an exact match, or a match up to a certain location in the mask. The Soundex Algorithm is also well known to those

skilled in the art, and will not be described in detail.

The classify program extracts all of the schema object names from the object oriented database in step 925 of **Figure 146**. As the names are extracted they are separated into distinct words in 926. Encoding of each of the distinct words into representative forms using the Bickel mask, and the Soundex mask occurs in 927. The process of steps 925 through 928 inclusive continues until all schema object names have been extracted, separated into distinct words and encoded. In addition to the extraction, separation, and encoding performed in 925, the program also remembers where in the schema tree structure that each word was used. Since the same word may be used in different locations of the tree, it is important to remember which one was used where.

User input is obtained in step 929. The user input is data which is obtained from a customer, and describes a part that is to be classified using the schema name data obtained in 925 through 928 inclusive. The user data is text data which can be broken into distinct words for the purposes of matching against schema words. Step 930 decomposes the user input string into distinct words. These words can be abbreviations of the intended words, can contain misspellings, or can be malformed in other ways.

The words obtained from the user description in step 930 are then encoded by both the Bickel algorithm, and the Soundex algorithm in step 931 of **Figure 147**. The Bickel character mask is applied to each schema word to determine which of the schema words are the best candidate matches in steps 932 through 935. Step 932 selects a schema word. Step 933 tests the score yielded by the Bickel Algorithm, to determine if it is the highest score match. If so, the result is saved in a list of potential matches in 934. The search continues until all schema words have been examined for matches in 935.

Step 936 in **Figure 148** examines the results of the previous search loop, and selects the words which exactly match the highest score seen by the loop from 932 to 935 from the list created of potential matches in 934. These soundex masks, created in 927, is used to test how well the refined set of candidates matches the original input word in 937. If a word

fails the Soundex test, then the word is discarded from the candidate list in 938. If the Soundex test succeeds in 937, the word is retained for further use, in 939, and the search continued to 940 to determine if any more candidate words remain. At 940 If there are still words in the candidate list, the program continues at 936. Otherwise, the program continues at 941.

At this point, there may still be too many options to be useful as a search tool, and only the use of single words have been discussed so far. The Description of parts in many cases involves multiple words. Each word bears meaning in itself, but collectively as a set of words mean more in this context than they do individually. In other words, the group is more meaningful than the sum of the component parts. This can be exploited by remembering where in the class structure each of the candidate words came from. By comparing the ancestry of the words that are found by the matching technique previously described, common threads can be found that are used more than other threads through the tree. These popular threads describe the likely places in the tree that a given part description can either be found , or could be placed. This process of finding common ancestor threads is performed in 941. **Figure 149** step 942 looks at the result of this process to determine if there are any words which did not gain any strength through the combination of ancestors. These words are discarded, unless they are the only ones. (In other words, no common ancestors were found.)

The output of these algorithms can then be presented to a user either through a text based interface, or a graphical user interface in step 943. The user can then make a selection from the small set to find what the user is looking for. This process is repeated for each description that is presented to the classify program either interactively, or through a batch interface.

The schema generator is used to generate object database structure from human entered text descriptions that contain abbreviations and spelling errors. There are three purposes of the schema generation program. The first is to generate schema class structure for an object oriented database from human generated text descriptions. The second is to determine the

class density of each of the generated classes. In other words, to determine how many user descriptions could be described by each generated class. The third is to combine erroneous spellings of the input data, to further populate existing classes, and to avoid creating additional variant classes.

The schema generation tool 3002 as shown in **Figure 199** is used to create schema using customer or user part descriptions as input. Since this part description data is entered by humans in 2150, the data tends to contain misspellings, and typographical errors. The schema generator reduces the user descriptions to schema structure in 2151. The output is a schema structure and a part mapping density that indicates how many of the part descriptions would be placed at each schema class in 2152.

The schema generator reads a description of an arbitrary part in step 960 of **Figure 150**. If a description was obtained, step 962 allows execution to continue at 963. If a input was not found, step 962 prints the output to a disk file and terminates . The part is decomposed into words in 963. Each word is compared against all other words that exist at that level in 964, to determine if this word needs to be added to the list at this level. If a the current word failed to match any previously used word, then the program proceeds to 971 of **Figure 151** to add the word to the internal word list at this level.

Regardless of how the match occurred (whether by adding a new word, or matching an existing word), then the next word in the string is examined against the subordinates of the matched word in a recursive fashion. This process continues until the input is exhausted, both in the case of each description, then until all descriptions are exhausted.

The techniques used for matching are the Bickel Algorithm in step 964 of **Figure 150,** a typographical error matcher in 968 of **Figure 151**, and an abbreviation matcher in step 966. In the Bickel Algorithm, a mask describes the characters which are common to the target word, and the candidate word is scored based upon the frequency of the use of each character. The higher the aggregate score, the better the match. This algorithm is used for finding the broadest set of potential matches and is used in 964 of **Figure 150**.

Since humans are not particularly consistent in the entering of part descriptions, additional techniques must be employed to refine the output of the search by the Bickel Algorithm. In some cases, humans use abbreviations in place of full word descriptions. Abbreviations are typicallly formed by either truncating the word, or deleting characters from the word.

In step 966 the abbreviation matcher attempts to stretch the target word when errors in comparison are encountered. Each time that a comparison fails, the target is stretched by 1 character. The comparison then resumes at the next character. If the target word gets longer than the candidate word, then the comparison fails. If the target is exhausted prior to completing the comparison with the candidate, then a match is declared if a certain percentage of the word has been covered by the suspected abbreviation. The required coverage is adjustable, and is tuned to each data set. Some examples would be:

```
Blt  -->B*lt
Bolt  Bolt            The comparison covers 100 %.

Rgstr       --> R*g*st*r
Register  Register        The comparison covers 100 %

Microproc       Microproc
Microprocessor Microprossor   The comparison covers 75
%.
```

The * characters here represent a character inserted into the string in the locations where the string is stretched. The value of the character is irrelevant to the process. Any character could be used. If the result of step 966 is to produce only 1 match, then the schema generator at step 967 will decide to combine the current word with a word that is currently in use in step 970

The typographical error matcher in step 968 is used to combine words which are intended to be the same, and can be detected by a human as the same word, but computers can not.

Typographical errors occur when a human entering data on a standard "qwerty" keyboard, misses the intended key, and instead uses one of the adjacent keys. An example would be the word

"Adhesive".

```
Adhesive  -- The intended word.
Ashesive  --  s used in place of the d.
Adnesive  --  n used in place of the h.
```

In both of these typos, the intended key can be found physically adjacent to the character that was used.

To match the words which contain typographical errors, two things must be done. A map must be made of all the adjacent keys for each character on the keyboard. Second, when comparing words which are suspected of containing typographical errors, the comparison must look at the keys adjacent to the character in the target word that does not match. If the key can be found as an adjacent key in the key map, then the comparison can continue. Each error is counted, and in the end, a match can be returned, with a particular number of errors. If there are still several candidates, the one with the fewest errors is selected.

In addition to the typographical errors described, there is a special case that involves transposed characters. This problem is not detected by the method described. However, by performing a subtraction on a character by character basis, and taking the absolute value of any number that is not zero, Transposed characters can be detected by adjacent characters which have the same non-zero difference value. Transpositions are not counted as errors for the purposes of error grading. An example would be the word "Positive".

```
Positive
Psoitive
```

When combined with the matching process that uses adjacent keys, the word Positive could be matched as follows:

```
Positive
Psoitice
```

If the result of the typo matching process in step 968 of **Figure 151** yields a single match, then the decision will be made in step 969 to combine the current part description with an

existing part description in step 970.

Once a matching has been deduced by the combination of these techniques, selecting the correct spelling of the word is the next problem. Each time that a new spelling is detected, and matched with a particular candidate word, the misspelled is combined with the candidate in step 970. If the candidate word is treated as an intelligent object, which can perform some actions when the replacement is made, then the number of times that any given spelling is observed can be recorded. The object can reflect its current state as the most popular spelling seen. Since humans tend to spell correctly most of the time, the misspellings still get matched, counted, and so forth, but they disappear in to the correctly spelled word. The object when asked what it spells responds with the most used spelling.

If a word is habitually misspelled by a particular user in their parts description, all of the misspellings congregate around the most popular one, which is suggested as a class entity in the object schema in one place. The misspelling can be corrected in one place rather than hundreds or thousands.

From 970, the program proceeds to 972 to determine if any more words remain in the current description. If there are more words, the program returns to 964 to begin evaluating the next word. This process continues for each word in a given description, and for each description, until all words and descriptions are exhausted.

Genic 3000 is a tool that is used as part of the legacy process. Genic 3000 is used for data augmentation and parameter specification for customer data that contains integrated circuits. The output of genic 3000 can be subsequently imported into a knowledge base 123 for additional legacy processing.

Figure 201 shows a typical data flow for processing data using Genic 3000.

Genic 3000 accomplishes data augmentation by using vendor part numbers and vendor names found in customer data shown in item 2162. The vendor part number and name is looked up programmatically in a published database of vendor parts as depicted by item 2160. Database 21620 and 2162 are read by genic 3000 in steps 2161 and 2163 respectively. In step 2164, information found in the published database is then translated

to ASCII text in a format 2165 that can be imported in step 2166 into a knowledge base 123.

The process for matching vendor part numbers found in customer data with published vendor part numbers may be a direct match or may involve several heuristics. If the part number does not directly match, the manufacturer name or code is needed and the vendor part number from the customer data must be decoded. The decoding is done by stripping off the prefix, suffix, and extracting a base device number.

The match algorithm scans the published database file, building a list of published candidate parts that match on at least the base number. In addition, matches of the manufacturer name, part number prefix, and part number suffix are noted. The number of different classifications of a base number (i.e. kinds of parts represented by the base number) is also determined. After these determinations have been made, the quality of the match can be determined.

The quality of a candidate match is based on a rating table shown in **Table 7**. The rating table is a Karnaugh map (see An Engineering Approach to Digital Design, William I. Fletcher, (1980), pg 134) reduction table shown in **Table 7**. The Karnaugh map is used in this case to guarantee that all possible combinations of the problem are considered, and to make it easier to express the relationships among the matches of manufacturer, number of classifications, prefix, and suffix. Each cell of the table contains a rating value.

The row and column indicies are boolean values indicating whether or not the value that is represented by that variable position is true or false. These boolean values indicate whether or not the associated condition is true which in turn indicates whether a match was made on this portion of the part number. The contents of the map are integer values representing the grade of that particular cell. In this map lower cell values (1) are preferred over larger cell values (12).

Table 7

Suffix/Manufacturer

Prefix / 1 Class	00	01	11	10
00	12	11	10	11
01	9	7	6	8
11	4	3	1	2
10	5	4	2	3

Table 7. A Karnaugh map which idenitifies relationships between match grades.

A table representation of the Karnaugh map in **Table 7** is described in **Table 8**. Table 8 is an english conversion table from match conditions to match grade. The grade of any given match can be determined by identifying which of the components match. It is assumed that the base number portion must match before any subsequent matching tests are made. Empty locations in the table indicate no match.

Table 8

Rating	Manufacturer Match	Suffix Match	Only One Class	Prefix Match
12				
11	Yes			
11		Yes		
10	Yes	Yes		
9			Yes	
7	Yes		Yes	
8		Yes	Yes	
6	Yes	Yes	Yes	
5				Yes
4	Yes			Yes
3		Yes		Yes
2	Yes	Yes		Yes
4			Yes	Yes
3	Yes		Yes	Yes

2		Yes	Yes	Yes
1	Yes	Yes	Yes	Yes

Beyond this representation, the table is reduced in yet a third representation programatically. In a program, the each match variable is assigned as a number, that when added to its peers, provides a unique index into an array of predefined values. The predefined values are the grade values themselves.

If numerical values are assigned, component match values are used for indexing grading for a match of each of the components, as follows:

Table 9

mnemonic	Value
Manufacturer	1
Suffix	2
1 Class	4
Prefix	8

The combination of these match values will yeild an index with the range of 0 - 15 inclusive. The contents of the grading array are shown in **Table 10** which is a lookup table which converts grade index to actual grade value.

Table 10

Index	Grade Value
0	12
1	11
2	11
3	10
4	9

5	7
6	8
7	6
8	5
9	4
10	3
11	2
12	4
13	3
14	2
15	1

The following is an example which works through this system, using a vendor (manufacturer) and a vendor (manufacturer) part number.

A list of candidate vendor part numbers for matching the vendor part number of Intel 2901B might look as follows:

Candidate #1: Manufacturer=AMD , Part Number = LM2901B
Candidate #2: Manufacturer=Intel , Part Number = 2901A

Assume that the base number 2901 was only found under one classification, microprocessors, in the published database.

Candidate #1 would be decoded to the example shown in **Figure 87**.

Using the match criteria described earlier, this part matches the Suffix, and was found in 1 Class. From Table 7, 8, or 10, this part match would be graded as an 8.

Similarly, candidate #2 would decode to the example shown in **Figure 131**

Again using the match criteria described earlier, this part matches the Manufacturer, the Prefix, and was found in 1 Class. In this case, the prefix was match, due to the absence of any prefix information in either description. From Table 7, 8, or 10 this part match would be graded as a 3.

Since the lower rating is better, the candidate #2 will be selected as a match for "Intel 2901B".

The output would contain parameter information from the

published database put into an import map and import file that maps to a knowledge base.

The operation of this software is described in the flow diagram **Figures 165 - 167**. Operation begins when the commercial database is read. The contents of this database is read, and indexed by base number in step 900 of **Figure 165**. After the database has been read, the program begins reading the input part data in step 901. In step 902, the part number that was received by 901 is decomposed into its component parts, namely a base number, a prefix, a suffix, and a manufacturer.

In step 903, the base number found in 902 is used to find matching database entries in the data read in 900. Item 904 determines if any base number matches were found in the commercial database data. If there were no base number matches, the program continues at 901 by looking for another vendor name and vendor part number as input. If one or more base number matches were found, the program continues at 905. At 905 the program proceeds for each of the matching entries found in 904, processing each matching item individually with a single pass through the loop for each matched item. In 905, one of the entries found in 903 is searched to determine if the item contains a match on the prefix portion of the part number that was identified in 902. If a match is found in 905, then the program continues at step 906 in **Figure 166** by setting the flag indicating a prefix match. The value of the prefix match is indicated in Table 8. If a prefix match was not found, then 906 gets skipped, and the program continues at 907. Item 907 searches the matching item being tested in this iteration , found in 903, for a match on the suffix portion found in 902. If a suffix match is found, operation continues at 908, by setting suffix match flag. The value of the suffix match flag is indicated in Table 8. If a suffix match is not found, program operation continues at 909.

In 909, the program searches the part being tested in this iteration, which was found in 903, for a match on the manufacturer portion found in 902. If a manufacturer match is found, then program operation continues at 910 where the match flag is set indicating a manufacturer match. The value of the manufacturer match is indicated in Table 8. If a manufacturer

match is not found, then 910 is skipped, and the program continues at 911. In 911, the program attempts to determine if all of the parts which matched the base number found in 902, and actually matched in 904 are of the same kind of part, or, in other words, do all of the parts found in 904 perform the same function. If they are determined to be the same, the program continues at step 912 of **Figure 167**. In step 912, the program sets the flag indicating that 1 class of parts was found for the part number requested in step 901. The value of the one class flag is indicated in Table 8. If the set of parts which matched in 904 are determined to represent multiple classes of parts, or parts which perform different functions, then the program continues at 913.

In 913 the program combines the match flags set in 906, 908, 910, and 912 into a single variable which constitutes the index into the grading table shown in Table 9. The index is used in 914 to lookup the grade in the internal array representation of Table 9. The grade found in 914 is assigned the part which matched in 904, and is the current part being used for this itteration, which began at 905. In 915, the program determines if there are more matches, which were found in 904, which need to be tested by this process. If more items remain, operation continues at 905 with the next item in the match list found in 903. If no more items remain to be examined, the program continues at 916. In 916, the program examines the results of the iterative process beginning at 905, and ending at 915 for the entries which have been scored the highest by 914. The highest grade is then selected by 916, and presented to the user as the best match. The best match contains all of the data associated with that entry. The data associated with that entry is obtained from the database which was read in 900.

To facilitate the legacy process, it is necessary automatically populate a knowledgebase with as much information necessary to uniquely identify a part and place the part in a class as close as possible to the part's actual class. This is achieved by importing customer data into a knowledgebase with a set of import utilities.

The import utilities may modify a knowledgebase in a number of ways. The most obvious way is by importing of new data into

the knowledgebase to create instances. Import utilities may add or modify data (parameters) on existing instances. Import utilities may also add enumerators that are missing from the schema, delete instances, translate text to other attribute types, and map unknown parameter values to known good values and numeric units, and dynamically change the destination class based information from a class map and customer data.

There are presently five import utilities: Import, Classmap Import, Simple Import, ImportA and ImportB. Though they are functionally similar, however, each one has unique features necessary to solve various scenarios associated with user data.

SimpleImport, Classmap Import and ImportB importing customer part information into a knowledgebase by creating new instances.

Import and ImportA allow importing part information into existing, selected instances, thus augmenting them.

Classmap Import dynamically changes the destination class through the use of a class map and a selected field in the user data.

Primary to the import utilities is the import file. The import file consists mainly of customer part data that is most useful for classifying parts. The import file must be formatted in a way acceptable to the import utilities. An import file has three sections: class path section, attribute name section and the customer data section.

The class path section is the first section in an import file. It is a single line of data, composed of tab separated names. The names are class names that specify the path from the root to the import destination class. Importing will generally be done at the destination class.

The attribute name section is the second section in an import file. It is a single line of data, composed of tab separated attribute names. The attribute names specify the attributes into which the customer data will be imported. The attribute names specified in this section must be valid at the destination class specified the class path section.

The customer data section is the third section in an import file. It is one or more lines composed of tab separated values. There is value for each attribute named in the attribute section. There is a one-to-one correspondence between the columns in the

attribute section and the columns of data in the customer data section. Two adjacent tabs represent an empty field.

The following figure illustrates the format of an import file. The first line is the class path section. The second line is the attribute name section. Lines 3 and 4 are the user data section. When imported, two instances will be created. The parameters for the Part Number and Description attributes will be set to 123321 and 1/4x11/2 20UNF for the first instance, and will be set to 123322 and 1/4x13/4 20UNF for the second instance.

```
root Mechanical Components    Fastners       Bolts
Part Number    Description
123321         1/4x11/2 20UNF
123322         1/4x13/4 20UNF
```

To allow modifying existing instances, certain attributes in the attribute section may be specified as "key" attributes. Key attributes are used to search and select certain instances. Only those instances whose parameters match those the values in the "key" attributes column in the import file are operated on during the import. Only string type attributes may be specified as key attributes. One or more attributes may be selected as key attributes. Key attributes may occur anywhere at any location in the attribute name section. An attribute name can be used both as a key name and as an attribute to be imported into. A key attribute is specified in the attribute section of the import file by prefixing the name with "key>" (e.g., key>Partno). The following figure illustrates an attribute name section of an import file that contains key attributes.

```
key>Part Number    Description    key>Vendor Code    V  e  n
Descript
```

To allow importing of numeric attributes a default unit must be specified in the attribute name section. A default unit is a unit name from the unit family associated to the attribute. Default unit specifier follows the "|" symbol, which is appended to the attribute name (e.g., Length|Inches). The following figure illustrates an attribute name section of an import file

that contains numeric attributes with unit specifiers.

Part Number	Description	Length|Inches	Diameter|Feet

Attribute values may be replicated to other attributes without adding an extra column of data to the customer data section in an import file. This is accomplished by specifying the two attribute names separated by a "!" symbol (e.g., Description!Description2). The following figure illustrates an attribute name section of an import file that contains an attribute that is replicated.

Part Number	Description!Description2

Comments may be placed in an import file by beginning a line with a "#" symbol. Blank lines are also allowed in an import file.

There are four phases performed during an import. After initialization, parsing command-line options, logging in, and opening the database, the first import phase is started.

The first phase is to read the first non-comment line of the import file step 1300 of **Figure 153**. This line is the class path section. The class path is read and parsed into class names. The class path is validated following the class path to the destination class. If the destination class exists, the second phase begins.

The second phase reads the second non-comment line of the import file (1302). This is the attribute name section. The line is read and the attribute names are parsed. The attribute names are validated by verifying that they exist at the destination class. For Simple Import and ImportA, all the attributes must be of type string. For Import, Classmap Import, and ImportB, all specified numeric attributes must also specify a valid default unit (special symbol "|"). In addition, attribute names are parsed for special other symbols like "key>" and "!".

The third phase only occurs when importing into existing instances. During this phase, a lookup table is created (1307).

The lookup table contains instance handle for each instance at the destination class and the parameter values for the key attributes (i.e., attributes prefixed by "key>" in the attribute name section). The lookup table is used to quickly locate instances that will be augmented by data in the customer data section of the import file.

The forth phase reads the data from the customer data section of the import file and performs the import (1308). This is done by reading each line, parsing the line into fields, and then importing the value by setting the parameter for the appropriate attribute.

In the case when the destination class is automatically selected, a selected field from the customer data is used to attempt matching a class from the class map file (1327). If a destination class can be identified, the new instance will be created in that class (1328) (1323).

In the case when new instances are being created, the instance is created prior to setting the parameters (1323).

In the case where data is being imported into existing instances, the key values are first extracted from the data, then a binary search is performed on the lookup table to identify all matching, existing instances (1324). Once all the matching instances are found, parameters are set from the remaining data.

In the case where enumerated attributes are imported step 1311 of **Figure 154**, if the attribute data from the import file does not match any existing enumerators associated to the attribute, then either the data is used automatically add a new enumerator to the schema, or the import utility will present a menu of existing enumerators to choose from (1317). If the menu is presented, the user may either choose map the data read from the file to an existing enumerator, or to use the data to add a new enumerator, or to ignore the data and leave the parameter undefined. If the user chooses to map the data to an existing enumerator, this information is retained by the import utility and is used if subsequent occurrences of the same data is encountered, at which time the utility automatically maps the data to the existing enumerator.

In the case where numeric attributes are imported (1312), if the attribute data from the import file is simply numeric

characters (1318), the parameter is set using the default unit for the attribute specified in the attribute name section of the import file. If the attribute data from the import file contains data that is both numeric and not numeric, it is assumed a unit specifier is included with the data and will be used to over-ride the default unit mentioned in the attribute name section. The utility will parse the data to location the unit specifier and see if the specifier names a known unit name (1319). If it does not, the import utility will present a menu of existing unit names to choose from (1320). When the menu is presented, the user may either choose map the data read from the file to an existing unit name or to ignore the data and leave the parameter undefined. If the user chooses to map the data to an existing unit name, this information is retained by the import utility and is used if subsequent occurrences of the same data is encountered, at which time the utility automatically maps the data to the existing unit name.

In the case where boolean attributes are imported (1310), if the attribute data from the import file is one of True, Yes, T, Y, or 1, the value of TRUE is assumed. If the data is one of False, No, N, F, or 0, the value of FALSE is assumed. If the attribute data from the import file contains data that cannot be recognized as either TRUE or FALSE (1314), the import utility will present a menu to choose from (1316). When the menu is presented, the user may either choose map the data read from the file to a TRUE value, or map the data to a FALSE value, or to ignore the data and leave the parameter undefined. If the user chooses to map the data to a TRUE or FALSE value, this information is retained by the import utility and is used if subsequent occurrences of the same data is encountered, at which time the utility automatically maps the data to the appropriate boolean value.

The following sections describes each of the five import utilities and how its features deviate from the general description of these utilities.

The usage and syntax of the import utility is shown in **Figure 125**.

Different operations may be performed depending on which options were set. If the -r option is set, the matching

instances are deleted from the knowledge base rather than imported. If the -M option is set, if no matching instances were found, a new instance is created from all the data on the line, including the key attribute fields. If the -U option is used, information is imported only when there is one matching instance. Nothing is imported if more that one instance is matched. If the -X option is used, information is imported only when there is no matching instance. This must be used with the -M option, otherwise there is a null effect.

Classmap Import adds new instances to a knowledgebase. Instances are added to a particular class based on a map that describes a pattern to match in data that class.

The usage of this command is shown in **Figure 126.**

After initialization the import map file is read. The import map file's first field is the pattern to match in the data and the second is the CADIS-PMX class to which an instance is imported. An exception file specified by the -o option is created. The exception file contains instances which could not be imported because the parameters did not match any of the map patterns.

The class path in the class path section of the import file, must name a class that all the attributes named in the attribute name section are valid.

When the customer data is read from the file, the field specified with the -f option is used to match patterns in the map file. If there is no match, the instance is output to the instance exception file specified with the -o option.

If the -f option field matches a pattern in the class map, the instance is added to that class and the parameters are set.

If an attribute for a parameter is an enumerated, the enumerator will be added to the schema if it does not exist This means that import must acquire a DBXLock.

If the replacement attribute for an instance parameter is a boolean, the parameter is set to TRUE if the text is an 'x', 'X', 'T', 't', 'TRUE', 'true', or a '1'. A FALSE value is set for '0', 'F', 'f', 'FALSE', or 'false'.

The format of an import map is shown below. For example, suppose the import were performed on the first data field on the first line in the import file which has a value of "10 inch

spike". That data field would be compared first against the pattern "Thyristor". This pattern does not match, so the data field is then compared against the pattern "*spike*" which contains regular expression meta-characters. This does match, so the first line would be imported to the class "Spikes" under the class "Mechanical" under the root "PMX_Root".

```
Thyristor PMX_Root  Electrical     Discrete  Thryistor
*spike*                    PMX_Root Mechanical     Spikes
```

Simple Import is the most basic import utility. It can only be used to create instances, and only string attributes may be specified. The command usage is shown in the following Table 11.

Table 11

simple_import [-u user] [-p password] -d kdbname [-P] [-v] import_file	
-u	use user name 'username' when doing login, if none given login id is used
-p	use specified password with login,
-d	the logical database name to connect to
-v	turn on verbose mode
import_file	name of the file containing the import info.
importA [-u user] [-p password] -d kdbname [-P] [-v]	No "key>" attributes allowed in attribute name section of import file. Only string type attributes may be imported. This utility is used to update existing instances if and only if the attribute's parameter is currently undefined.

import_file

-u	use user name 'username' when doing login, if none given login id is used
-p	use specified password with login,
-d	the logical database name to connect to
-v	turn on verbose mode
import_file	name of the file containing the import info.

This utility is used to create new instances and allows the importation of all attribute types.

```
importB [-u user] [ -p password ] -d kdbname [-P] [-v] -o nogoparts import_file
```

-u	use user name 'username' when doing login, if none given login id is used
-p	use specified password with login,
-d	the logical database name to connect to
-v	turn on verbose mode
-o	the name of the file to write part data of parts that could not be imported due to difficulty with setting parameters.
import_file	name of the file containing the import info.

II. Additional Embodiments and Modifications

Although the invention has been described herein with reference to an application to the problem of parts management, those skilled in the art, after having the benefit of this disclosure, will appreciate that the invention is useful in other applications as well. For example, the invention will be particularly useful in any application where an organization places value on reliably finding one of many instances of objects having variable descriptions. The dynamic class manager

described herein will be particularly useful in any application where it is desirable to restructure a classification or schema.

Although the invention has been described herein with reference to a local area network, those skilled in the art, after having the benefit of this disclosure, will appreciate that other embodiments and implementations are possible. For example, the system could be implemented on a main frame or single computer having multiple user stations. The system could also be implemented over a network other than a LAN, such as a wide area network or the InterNet.

Additional file manager 140 derivations are possible. The interface provided by the file manager 140 to the dynamic class manager 134 and the handle manager 137 is an agreement to maintain a copy of the dynamic class manager schema and instance data on secondary persistent storage 103. Changes, as they are made to the schema and instances are also made in secondary storage. The dynamic class manager 134 is initialized by reading the data, via the file manager 140, from secondary storage 103. Other secondary storage mechanisms could be implemented which follow the interface specification. Other implementations could use commercial data bases including relational database management systems such as an Informix database, Oracle database, Raima database, etc. Other implementations could also be built using other proprietary file formats.

III. Method and Apparatus for Concurrency in an Object Oriented Database

A. Overall Architecture

A presently preferred embodiment of the present invention is shown in **Figure 204**, and employs a network 4100 having a client/server architecture comprising one or more knowledge base clients 4112, 4118 and 4111, and a knowledge base server 4108. In the preferred embodiment shown in **Figure 205**, the knowledge base server 4108 includes an object oriented lock manager 4125, a dynamic class manager 4134, a connection manager 4135, a query manager 4136, a handle manager 4137, a units manager 4138, a database manager 4139, and a file manager 4140. A server host 4109 may be designated to run the knowledge base server 4108, with the software and knowledge base 4123 preferably residing on a local disk drive 4110. A knowledge base client 4131 interacts

with the knowledge server 4132 over a network 4100 in the illustrated embodiment. A preferred system includes a registry server 4141 and a license manager 4142 to control unauthorized access to the system. A legacy client 4133 and a legacy manager 4145 are preferably included to facilitate organization of an existing legacy database into schema for use in connection with an object oriented database management system. An application programming interface or API 4143 is also shown in the illustrated embodiment.

A schema editor 4144 is provided for modifying and changing the schema or database 4123. With the concurrency control provided by the present invention, a plurality of schema editors 4144 may be used at the same time, while a plurality of retrievers 4130 are being used. The structure and operation of the schema editor 4144, the dynamic class manager 4134, the retriever 4130, the connection manager 4135, the query manager 4136, the handle manager 4137, the units manager 4138, the database manager 4139, the file manager 4140, the registry server 4141, the license manager 4142, the API 4143, the legacy manager 4145, and the knowledge base client 4131 are described above in more detail.

B. Concurrency Control

In the example illustrated in **Figure 204**, a plurality of users or clients 4111, 4112, and 4118 are shown connected to the network 4100. A first client 4111 runs on a Sun Microsystems SPARCstation 4111, which is shown having a display 4116, a mouse 4117, and a keyboard 4122. A second client 4112 runs on an IBM compatible computer 4112, shown having a display 4113, a mouse 4114, and a keyboard 4115. A third X Windows client 4118 is illustrated having a computer 4118, a display 4119, a mouse 4120, and a keyboard 4122.

The present system supports interactive editing by one or more users connected through respective clients 4131. For example, users are able to use a schema editor 4144 to change the schema by adding and deleting attributes, to add whole sections to the schema, to reposition whole sections of the schema within the schema hierarchy, and to modify, add and delete instances. These interactive editing operations may be performed while other users are simultaneously using retrievers 4130 to access the

database 4123. The management of these simultaneous operations, including the ability of a plurality of users to access the database 4123 while a plurality of users are at the same time making changes to the database 4123, is referred to as concurrency control.

In the present invention, the object oriented lock manager 4125 provides a concurrency control mechanism which allows users to query and view class objects without disruption of their view while modifications are being made by other users. These modifications include additions, deletions, and edits of classes, attributes, instances, and parameters.

In a preferred embodiment, the lock manager 4125 is a subsystem of the class manager 4134.

The present invention optimizes performance of the concurrency control system by using lock inheritance based on class objects. The lock manager 4125 implements a mechanism for locks to be placed on a class without subclass inheritance of the lock. This mechanism is referred to as a "class lock." The lock manager 4125 also provides an inheritance mechanism for locks. The inheritance mechanism is referred to as a "tree lock." Tree locking a class will effectively result in a "lock" on all descendants of that class by inheritance without physically requiring the placement of class locks on the descendant classes.

The present invention simplifies the number of objects that need to be locked by using class level lock granularity. This optimizes performance. The granularity or scope of a class lock is the class itself, the attributes defined by the class, and the instances associated with that class. Figure 206C is a schematic diagram that depicts a hierarchy of lock granules in accordance with the present invention. A significant feature of the present invention is that it does not allow an instance to be locked independently of the class to which it belongs. This is in contrast to the approaches shown in **Figure 206A** and **Figure 206B**. In the present invention, classes are locked, either individually (class locks), or in groups (tree locks), but instances are not locked as such. Concurrency is controlled, not by determining whether a instance in question is itself locked, but rather by determining whether the class to which it belongs is locked. The composite object is a class.

The present invention can implement concurrency control in an object oriented database using only three types of lock modes, although four types are preferably employed. The three types of lock modes used in the present invention are: class share lock ("CSL"), tree update lock ("TUL"), and tree exclusive lock ("TXL"). The fourth type of lock mode that may be used is a tree share lock ("TSL"), which may be considered to be in effect a group of class share locks. Therefore, in a preferred embodiment, the knowledge base server 4132 actually supports four lock types: exclusive, update, and two flavors of share locks.

The "class share lock," which is also referred to as a "CSL," locks a single class node for sharing.

The "tree share lock," which is also referred to as a "TSL," locks the subtree rooted at the class for sharing. This lock behaves exactly like placing a CSL on each class in the subtree.

The "tree update lock," which is also referred to as a "TUL," locks the subtree rooted at the class for instance editing. This is sometimes called simply an 'update lock' or U-lock.

The "tree exclusive lock," which is also referred to as a "TXL," or sometimes simply as an X-lock, locks the subtree rooted at the class for exclusive use.

Some actions which change the knowledge base 4123 can be performed without requiring an exclusive type of write lock. Another type of write lock, referred to herein as an "update" lock, is used for certain actions including modifying parameter values, adding, and moving instances. An update lock is a hybrid of the share and exclusive locks. An object may be update locked by at most one application, but simultaneously the object can be share locked by one or more applications. This means that the one application with the update lock can make changes to the object at the same time as it is being examined by the other applications. These changes to the knowledge base that can occur when an object is both update and share locked are considered easy enough for an application to recognize and manage.

An update lock is a "weaker" type of a write lock than an exclusive lock. Any change to the knowledge base 4123 requires that a write lock has been requested and acquired. Some of the updating actions require an exclusive lock, and other updating actions require an update lock. But, the ones that require an

update lock require "at least" an update lock. An exclusive lock is always sufficient for making a change to the knowledge base 123, but an update lock is a more friendly, more concurrent lock for making a selected set of changes.

The knowledge base client 4131 uses the object oriented lock mechanisms provided by the lock manager 4125 to place locks of appropriate granularity and inheritance to provide the maximum availability, stability, and performance of a tool using these mechanisms. The example described herein is optimized for a read oriented database system. It is particularly advantageous in a knowledge base schema that is used for parts management.

Locks serve two purposes. First, locks are used by the application or knowledge base client 4131 to announce or make the statement that an object is being examined. Since it is harmless for multiple applications to examine the same object simultaneously, the type of lock used for this purpose is a share lock. Several applications can share an object by concurrently share locking it. Typically, applications use share locks as they navigate through the schema, perform queries, and examine instances.

The second use of locks by an application is to announce that it wishes to change an object. The application should insure that no other application is attempting to change the same object. This type of lock is called an write lock. Other applications are prevented from changing an object that is write locked. Typically, applications use write locks when adding or deleting instances, modifying parameter values, or editing the schema. As noted above, the knowledge base server 4132 supports two types of write locks: exclusive locks and update locks. Exclusive locks are used to prevent applications from interacting in ways that could cause problems. For example, when an instance is to be deleted, or when the schema is edited, an exclusive lock is used. Where an object can be changed in ways that do not cause problems, a weaker update lock is preferably used to provide maximum concurrency.

It will be appreciated that most of the locks used in the present invention are 'tree' locks. In the above discussion, references were made to locking an object (actually a class). What is really meant is that a class is under the influence of

a lock. When the ancestor class of a given class is exclusive locked, then that class is also effectively exclusive locked because it is in the subtree which is rooted by an exclusive locked class.

An application establishes a lock by requesting it. If the request succeeds, then the application has acquired the lock. The application must release the lock when the object no longer needs to be locked. The request will fail if the lock conflicts with other locks that have already been requested by other applications. A conflict will occur if the request is for a write lock and the object is already write locked or if the request is for a share lock and the object is already exclusive locked.

The objects that can be locked are always classes. Instances are never locked. The preferred system uses a subtree as an alias for an instance. In this approach, fewer locks are applied, which results in a less complex and faster system. For an application to change some object which is not a class, a write lock on the class associated with that object is required. In other words, to add an instance a write lock must be requested for the class to which the instance is to be added. A parameter value can only be changed when the application requests a write lock on the class that owns in instance. For example, the schema developer or editor 4144 requests exclusive locks on a class for making changes to attributes which are defined by that class.

The lock manager 4125 and the knowledge base server 4132 require an application to become a lock holder before it can request a lock. It becomes a lock holder by using the pmx_startLockHolder() function, and thus starting a lock holder. The pmx_startLockHolder() function is described more fully in the software functions section. The combination of the application's connection to the knowledge base server 132 and the lock holder are what distinguish one application from another for resolving conflicts between locks. An application can start multiple lock holders and thus cause conflicts for lock requests within the application. This is useful for subroutines within the application that need to be isolated. The application stops being a lock holder by ending the lock holder.

Each application connection to the knowledge base server has

a unique lock holder table 4146 as shown in Figure 205. The lock holder table 4146 is used by the lock manager 4125 to manage the existing lock holders for each connection.

Figure 255 shows the data structure for the lock holder table 4146. In a preferred embodiment the lock holder table 4146 is a dynamic list of Boolean values.

A TRUE value in the lock holder table 4146 represents a lock holder that has been started. A FALSE value in the lock holder table 4146 is a lock holder that has been ended or one that has never been used. The index into the lock holder table 4146 is the lock holder handle 267 itself. Thus, in the example shown in Figure 255, the TRUE value 4601 is lock holder handle zero, and it has been started. The lock holder handle one 4267 corresponds to the table 4146 entry identified by reference numeral 4602, and it has a TRUE value indicating that it has been started. The lock holder handle 2 with value FALSE 4603 has been ended.

The operation of starting a lock holder is shown in the flow chart in Figure 256. In step 4607, the lock holder table 4146 is searched for a value of FALSE, representing a lock holder that is not in use and can be allocated. If a FALSE element is found, then the table index is assigned to "newLH." In step 4608, if a FALSE element was found control proceeds to step 4609 where the the lock holder table 4146 element at index "newLH" is set to TRUE to indicate that the lock holder is being allocated. If a FALSE element was not found in step 4608, control continues at step 4611 where a new element 4606 is allocated at the end of the lock holder table 4146 and the index of this new element 4606 is assigned to "newLH". Control continues at step 4609. At step 4610, the index "newLH" is returned as the newly started lock holder handle.

Figure 257 is a flow chart for the operation of ending a lock holder. The process can be performed very quickly in one step 4612 in the present invention. In step 4612, the lock holder table 4146 element indexed by the lock holder handle to be ended is set to FALSE.

Figure 208 is a diagram representing the lock conflicts for the lock types and granularities provided in the present invention. The first column 4220 represents locks held by a

first user, who will be referred to as lock holder 1. The top row 4219 represents the lock requested by a second user, who will be referred to as lock holder 2. The conflict states are shown in the intersecting cells. The cells indicate whether the lock requested by lock holder 2 conflicts with the lock held by lock holder 1. For example, if lock holder 1 has a TUL on a class, represented by the location in column 4220 indicated by reference numeral 4216, and lock holder 2 requests a CSL, represented by the location in row 4219 indicated by reference numeral 4217, then the intersecting cell 4221 shows that there is no lock conflict and lock holder 2 gets the CSL on the class.

Table 12 lists the available lock types used by the present invention, lock granularities and their mnemonics. The most restrictive locking mechanism is the exclusive lock which only allows one lock holder. The most permissive lock type is a share lock which allows multiple lock holders of non-conflicting types. An intermediate level of concurrency is provided by the update lock. Although an object oriented lock manager may provide class exclusive locks or class update locks, the tree granularity for the lock types used in the preferred embodiment of the present invention are sufficient to provide view stability. Share locks are preferably provided at both the class and tree granularity, but that is not required by the present invention.

In a preferred embodiment, concurrency control primarily occurs at the application level, and not at the DBM (database management) level. The client application 4130, 4144 or 4133 of the API 4143 must explicitly request a lock when the application attempts a function. Although the description herein sometimes refers to a user or lock holder "requesting" a lock, in a preferred embodiment, the GUI programs may be written so that a user does not need to explicitly perform such a request in the sense that the GUI programs hide this operation from the user and the user may not actually be aware that the client application 4130, 4144, or 4133 of the API 4143 is requesting a lock. The client application may perform a background request to the lock manager 4125 when the user attempts to navigate the hierarchy or edit parts, for example using the retriever 4130 or the schema editor 4144. If a conflict is detected or the request fails, the user is then informed through an appropriate dialog box or

message that access to the portion of the schema that the user attempted to navigate or edit cannot be obtained. In a preferred system, the client applications 4130, 4144 and 4133 are well behaved and cooperate to achieve concurrency control. In other words, concurrency is mediated by cooperating applications 4130, 4133, and 4144.

A given application and lock holder combination can request multiple locks of the same type for the same class without conflict. For example, in the above description with reference to Figure 208, the same user could be both lock holder 4001 and lock holder 4002. This may occur, for example, when the same user opens a second window. A count for each type of lock acquired by the application is maintained by the lock manager 4125 of the knowledge base server 4132. The locks must be released as many times as they are requested. However, in a preferred embodiment, locks can be released en masse in five ways. The knowledge base server supports two API functions for releasing multiple locks. All locks that have been acquired by a lock holder are released when the lock holder is ended. And, all locks that have been acquired by an application are released when the application closes the knowledge base 4123 or when the application logs out.

The share locks supported by the lock manager 4125 of the knowledge base server 4132 are advisory. This means that the share lock is a means of announcing to other applications (ones that might want to edit instances or the schema) that a part of the schema is being navigated. Share locks are not required for navigating the schema or for querying and examining instances, but they are preferred. Acquiring a share lock prevents other applications from acquiring write locks, which are enforced. The lock manager 4125 and the knowledge base server 4132 will not allow any schema or instances to be edited without appropriate write locks. Therefore, if clients of the API 4143, such as the retriever 4130, schema editor 4144, legacy 4133, or user written API program, requests share locks whenever one of them navigates into a part of the schema, it will be insulated from any changes that might occur to the schema while it is navigating.

The client application 4130, 4144, and 4133 of the API 4143 should request a class share lock for a class whenever it gets a class descriptor or attribute descriptor for that class. This

method insures that the data in the descriptor is valid and remains valid. The client application 4130, 4144, and 4133 should also use a class tree lock at a class for which it does a query. This may be used to prevent another application from, for example, deleting instances in the subtree where the query is applied.

In the present invention, locks are not subsumed. An object may have multiple locks of the same type. Lock requests and releases are paired. In the illustrated embodiment, a function to perform a release of a class share lock will only release one class share lock on an object.

The operation of the lock manager 4125 may be better understood with reference to Figures 209-211. Figure 209 is a schematic diagram of a class hierarchy 4215 representing an example of a portion of an object oriented database. In this example, class 4202 is an ancestor for all of the other classes which are depicted. If Figure 209 depicted the entire database, then class 4202 would be the root class. Class 4202 is the parent of class 4201 and class 4205. Class 4201 has two children shown as class 4206 and class 4200. Class 4205 is the parent of class 4210 and class 4207. Class 4200 has two descendants: class 4203 and class 4204. Class 4206 has two children shown as class 4208 and class 4209. Similarly, class 4210 and class 4207 each are shown with two children: classes 4211 and 4212, and classes 4213 and 4214, respectively.

If a lock is requested for class 4200, the first step is checking whether the requested lock conflicts with any other lock at this class 4200. This is represented in Figure 209, where class 4200 is shown as a black square to represent the step of examining the class 4200 for conflicting locks at this point in the hierarchy 4215. The determination of conflicts is performed in accordance with the matrix represented in Figure 208. If the requested lock for class 4200 is a class share lock CSL, and the class 4200 is already subject to a class share lock CSL, a tree share lock TSL, or a tree update lock TUL, then there is no conflict, and the answer "No" (i.e., no conflict) would be returned. This is represented in Figure 208 as a "No" at the intersection of the CSL column with the CSL, TSL, and TUL rows. If the requested lock for class 4200 is a class share lock CSL,

and the class 4200 is already subject to a tree exclusive lock TXL, then there is a conflict and the answer "Yes" (i.e., yes there is a conflict) would be returned. This is represented in Figure 5 as a "Yes" at the intersection of the CSL column with the TXL row. If there is a conflict, the requested lock is not granted.

The lock request procedure would then continue in this particular example with the step of checking whether the requested lock conflicts with any other lock at the ancestors 4201 and 4202 of the class 4200. This is represented in Figure 210, where class 4201 and class 4202 are shown as a black squares to represent the step of examining the ancestor classes 4201 and 4202 for conflicting locks at these points in the hierarchy 4215. The determination of conflicts is performed in accordance with the matrix represented in Figure 208. The class 4200 is represented in Figure 210 as a shaded square to indicate that the class 4200 is the class for which the lock is requested. After the check in Figure 6 is completed successfully, the ancestor classes 4201 and 4202 of class 4200 are checked for conflicts. In this example, the request for a lock on class 4200 could be for either a class or tree lock. If a conflict is indicated, the requested lock is not granted. If no conflict is detected, the answer "No" is returned. In such a case, the requested lock may be granted if the requested lock is a class share lock. If the requested lock is a tree exclusive lock, a tree share lock, or a tree update lock, the procedure continues to the step described in connection with Figure 211.

Figure 211 is a diagram illustrating a hierarchy during a subsequent step in the process of granting a tree lock request on class 4200, if the checks in Figure 209 and Figure 210 are successful. The descendent classes 4203 and 4204 are checked for conflicts. The class 4203 and the class 4204 are each shown as a black square to represent the step of examining the descendent classes 4203 and 4204 for conflicting locks at these points in the hierarchy 4215. The determination of conflicts is performed in accordance with the matrix represented in Figure 208. The class 4200 is represented in Figure 211 as a shaded square to indicate that the class 4200 is the class for which the lock is requested. If a conflict is indicated, the answer "Yes" is

returned and the requested lock is not granted. If no conflict is detected, the answer "No" is returned and the requested lock is granted.

The operation of the lock manager 4125 may be best understood with reference to Figures 247-254. During operation, the lock manager 4125 maintains a dynamic lock table 4400 shown in Figure 254. The lock table 4400 interacts with the schema. For example, if a class is physically added or deleted from the schema, the lock table 4400 is changed accordingly. Locks are evaluated by the system based upon the inheritance pattern reflected by the schema. The lock table 4400 is maintained in the illustrated example by the knowledge base server 4132.

The lock table 4400 shown in Figure 254 is organized in the preferred embodiment so that each row corresponds to a class in the schema. Each column corresponds to a lock holder using the system. Each cell of the lock table 4400 has been numbered for purposes of reference during the discussion that follows. For example, the intersection of the row corresponding to class handle 4003 and the column corresponding to lock holder 4002 is indicated by reference numeral 4410. If the a class share lock is placed on the class corresponding to class handle 4005 by the user corresponding to lock holder 4003, then the lock manager 4125 would place a CSL indication in element 4419 of the lock table 4400. It will be appreciated by those skilled in the art that there is no provision in the concurrency control system according to the present invention for locking an instance; the lock table 4400 only makes provision for classes.

If lock holder 4006 attempted to place some type of lock on the class corresponding to class handle 4004, the lock manager 4125 would have to check element 4404 of the lock table 4400 to determine whether lock holder 4001 had a conflicting lock on that class. The determination of what lock type conflicts with the type of lock that the lock holder 4006 was attempting to place on the class would be determined in accordance with the lock conflict table of Figure 208. If no conflicting lock was found at cell 4404, then the lock manager 4125 would proceed to check cell 4411 to determine whether lock holder 4002 had a conflicting lock on the class corresponding to class handle 4004. If not, the lock manager 4125 would proceed to check cell 4418 to

determine whether lock holder 4003 had a conflicting lock on the class. The lock manager 4125 would continue until all corresponding cells 4425, 4432 and 4446 for the remaining lock holders 4004, 4005, and 4007, respectively, were checked. This is essentially the procedure corresponding to the process represented in Figure 209.

In order to perform checks of ancestor classes, for example checking class 4201 shown in Figure 210, the lock manager 4125 must have a mechanism to supply the lock manager 4125 with information as to what class handle corresponds to the class 4201. The dynamic class manager 4134 performs this function. Thus, in order to implement the ancestor check depicted in Figure 210, the dynamic class manager 4134 will supply the lock manager 4125 with the class handle for the ancestor class 4201. If the corresponding class handle is class handle 4002, then the lock manager 4125 can perform a check of the cells 4402, 4409, 4416, 4423, 4430, and 4444 in the row corresponding to the class handle 4002 in the manner described above with reference to the row for class handle 4004.

Similarly, in order to perform a check of descendent classes 4203 and 4204 shown in Figure 211, the dynamic class manager 1434 will supply the lock manager with the class handles corresponding to these classes, and the lock manager may check the corresponding rows of the lock table 4400 to determine if there is a conflicting lock. When an operation involves an instance, the dynamic class manager 4134 supplies the lock manager 4125 with the owning class for that instance, and the system checks for lock conflicts with that class.

When a lock is requested, the lock manager uses both the connection and the lock holder handle 4267 for identifying lock conflicts. When a schema or instance edit is attempted, the dynamic class manager 4134 first asks for authorization to perform the operation from the lock manager 4125. In one embodiment, only the connection is used to check for authorization. In this example, the lock holder that asked for the edit operation is not taken into account when checking for the existence of the appropriate lock. This optimization was done in this particular example to prevent requiring a lock holder handle as an input argument to each of the API editing

functions.

Figure 247 is a flow chart depicting the steps for requesting authorization to do a schema edit. An exclusive lock is required by the lock holder in order to do the desired schema edit. In step 4450, the current class is set equal to the class to be checked. In step 4451, the "current class" is checked to see if it is exclusive locked, (i.e., whether it has a tree exclusive lock TXL). Referring to Figure 254, if the requesting lock holder is lock holder 4003, and the current class is class handle 4003, this step in effect checks the intersection cell 4417 for an exclusive lock. If it is exclusive locked, then it means in this example that it is exclusive locked by the lock holder that is attempting to do the edit. In that event, the lock manager 4125 returns an "OK" indication in step 4452 to the client 4131 corresponding to the requesting lock holder 4003. If it is not exclusive locked, the flow proceeds to step 4453 where the lock manager 4125 checks to determine whether the "current class" is the root class. If it is the root class, the lock manager 4125 returns a "no" in step 4454. If it is not, the flow proceeds to step 4455, where the "current class" is set equal to the parent class of the class that was the "current class." The lock manager 4125 asks the dynamic class manager 4134 who the parent is, and that information is supplied to the lock manager 4125 by the class manager 4134. The procedure then loops back to step 4451, as shown in Figure 247. In effect, the lock manager 4125 will check the ancestors using this procedure.

Figure 248 is a flow chart depicting the steps for requesting authorization to do an instance edit. In order to perform an edit of an instance, an exclusive lock or an update lock is required. The lock manager 4125 must first ask the class manager 4134 to tell the lock manager 4125 who is the owning class for the instance, and this information is provided by the dynamic class manager 4134. In step 4457, the current class is set equal to the class to be checked. In step 4458, the "current class" is checked to see if it is exclusive locked or update locked, (i.e., whether it has a tree exclusive lock TXL or a tree update lock TUL). If it is exclusive or update locked, then it means in this example that it is so locked by the lock holder that is doing the edit. In that event, the lock manager 4125 returns an

"OK" indication in step 4459 to the client 4131 corresponding to the requesting lock holder. If it is not exclusive locked, the flow proceeds to step 4460 where the lock manager 4125 checks to determine whether the "current class" is the root class. If it is the root class, the lock manager 4125 returns a "no" in step 4461. If it is not, the flow proceeds to step 4462, where the "current class" is set equal to the parent class of the class that was the "current class." The lock manager 4125 asks the dynamic class manager 4134 who the parent is, and that information is supplied to the lock manager 4125 by the class manager 4134. The procedure then loops back to step 4458, as shown in Figure 248.

Figure 249 is a flow chart depicting the steps for requesting a class share lock. In step 4464, the "current class" is set equal to the class for which the lock is requested. In step 4465, the class is checked to determine whether it is exclusive locked by some other lock holder. If it is, the lock manager 4125 returns a "no" in step 4466. If it is not, the lock manager 4125 proceeds to step 4467, where the lock manager 4125 checks to determine whether the "current class" is the root class. If it is, the lock manager 4125 returns a "yes" and grants the requested CSL in step 4468. If it is not, the lock manager 4125 proceeds to step 4469, where the lock manager 4125 asks the class manager who the parent class is. When that information is supplied to the lock manager 4125, the "current class" is set equal to the parent class, and the flow loops back to step 4465.

Figure 250 is a flow chart depicting the steps for requesting a tree share lock. In step 4470, the "current class" is set equal to the class at which the tree lock is requested. In step 4471, the "current class" is checked to determine whether it is exclusive locked by some other lock holder. This checks the row in the lock table 4400 corresponding to the "current class" at every cell except the cell in the column corresponding to the requesting lock holder. If it is, the lock manager 4125 returns a "no" in step 4472. If it is not, the lock manager 4125 proceeds to step 4473, where the lock manager 4125 checks to determine whether the "current class" is the root class. If it is not, the lock manager 4125 proceeds to step 4474, where the lock manager 4125 sets the "current class" equal to the parent

class (the lock manager 4125 must obtain the identification of the parent class from the class manager 4134). The procedure then loops back to step 4471. This effectively results in checking the ancestors. If it is found to be the root class in step 4473, the lock manager 4125 checks to see if all of the descendent classes have been checked in step 4475. If they have, then the lock manager 4125 returns a "yes" and grants the requested TSL in step 4476. If not, in step 4477 the lock manager 4125 sets the "current class" equal to some descendent that has not yet been examined.

In step 4478, the lock manager 4125 then checks to determine whether the new "current class" is exclusive locked by some other lock holder. This effectively results in checking the corresponding row in the lock table 4400 at every cell except the cell in the column corresponding to the requesting lock holder. If the new "current class" is not exclusive locked by some other lock holder, the flow loops back to step 4475. This loop effectively results in checking all of the descendants. If the new "current class" is exclusive locked by some other lock holder, then the lock manager 4125 returns a "no" in step 4479.

Figure 251 is a flow chart depicting the steps for requesting a tree update lock. In step 4480, the "current class" is set equal to the class at which the tree lock is requested. In step 4481, the "current class" is checked to determine whether it is exclusive locked or update locked by some other lock holder. This checks the row in the lock table 4400 corresponding to the "current class" at every cell except the cell in the column corresponding to the requesting lock holder. If it is, the lock manager 4125 returns a "no" in step 4482. If it is not, the lock manager 4125 proceeds to step 4483, where the lock manager 4125 checks to determine whether the "current class" is the root class. If it is not, the lock manager 4125 proceeds to step 4484, where the lock manager 4125 sets the "current class" equal to the parent class (the lock manager 4125 must obtain the identification of the parent class from the class manager 4134). The procedure then loops back to step 4481. This effectively results in checking the ancestors. If it is found to be the root class in step 4483, the lock manager 4125 checks to see if all of the descendent classes have been checked in step 4485. If

they have, then the lock manager 4125 returns a "yes" and grants the requested TUL in step 4486. If not, in step 4487 the lock manager 4125 sets the "current class" equal to some descendent that has not yet been examined.

The lock manager 4125 then checks in step 4488 to determine whether the new "current class" is exclusive locked or update locked by some other lock holder. This effectively results in checking the corresponding row in the lock table 4400 for the new "current class" at every cell except the cell in the column corresponding to the requesting lock holder. If the new "current class" is not exclusive locked by some other lock holder, the flow loops back to step 4485. This loop effectively results in checking all of the descendants. If the new "current class" is exclusive locked or update locked by some other lock holder, then the lock manager 4125 returns a "no" in step 4489.

Figure 252 is a flow chart depicting the steps for requesting a tree exclusive lock. In step 4490, the "current class" is set equal to the class at which the tree lock is requested. In step 4491, the "current class" is checked to determine whether it has any lock by some other lock holder. This checks the row in the lock table 4400 corresponding to the "current class" at every cell except the cell in the column corresponding to the requesting lock holder. If any other lock holder has any type of lock on the "current class," the lock manager 4125 returns a "no" in step 4492. If it is not, the lock manager 4125 proceeds to step 4493, where the lock manager 4125 checks to determine whether the "current class" is the root class. If it is not, the lock manager 4125 proceeds to step 4494, where the lock manager 4125 sets the "current class" equal to the parent class (the lock manager 4125 must obtain the identification of the parent class from the class manager 4134). The procedure then proceeds to step 4495, where the lock manager 4125 checks to determine whether the new "current class" has a TSL, TUL or TXL by any other lock holder. If it does, the lock manager 4125 returns a "no" in step 4496. If it does not, the flow loops back to step 4493. In step 4493, if the "current class" is found to be the root class, the lock manager 4125 checks in step 4497 to see if all of the descendent classes have been checked. If they have, then the lock manager 4125 returns a "yes" and grants the

requested TXL in step 4498. If not, in step 4499 the lock manager 4125 sets the "current class" equal to some descendent that has not yet been examined.

The lock manager 4125 then checks in step 4500 to determine whether the new "current class" is has any type of lock by some other lock holder. This effectively results in checking the corresponding row in the lock table 4400 for the new "current class" at every cell except the cell in the column corresponding to the requesting lock holder. If the new "current class" does not have any type of lock by some other lock holder, the flow loops back to step 4497. This loop effectively results in checking all of the descendants. If the new "current class" has any type of lock by some other lock holder, then the lock manager 4125 returns a "no" in step 4501.

When a client 4131 invokes a retriever 4130, the concurrency system will perform the procedure depicted in Figure 212 to open a retriever window 4290. Figure 212 is a flow diagram representing the locking process performed when the retriever window 4290 is opened. In step 4225, the user attempts to open a retriever window 4290. A new lock holder is requested in step 4226. If the request for a new lock holder in step 4226 fails, then the flow proceeds to step 4227, and the client 4131 will not display a retriever window. If the request for a new lock holder is granted, the flow proceeds to step 4228.

The new lock holder is associated with that user. In many cases there may be a one-to-one correspondence between users and lock holders. However, a single user can be more than one lock holder, so the following discussion will refer to lock holders. In the procedure depicted in Figure 212, the new lock holder then requests a CSL for the root class in step 4228. In the illustrated example, a GUI associated with the client 4131 will request the class share lock for the root class. If the requested CSL is not granted, the flow proceeds to step 4227, and the retriever window will not be displayed. Preferably, a message to the user is generated by the system in step 4227. If the CSL requested in step 4228 is granted, the flow proceeds to step 4229, and a retriever window is opened for the lock holder and displayed on the user's display 4116.

Figure 213 depicts a process 4230 that is performed by the

system when a class is selected in the class hierarchy. When the user attempts to select the class in step 4232, a request for a CSL is issued in step 4233 by the GUI associated with that user's knowledge base client 4131. If the request fails, the flow proceeds to step 4234; the class is not selected. If the CSL is granted, the flow method proceeds to step 4235, where the class becomes the selected class, becomes highlighted, and associated attributes are displayed.

Figure 216 shows an example of a screen that may be displayed on the user's display 4116 when the user is navigating the class tree 4248. A root class 4245 is designated class 4001 in the display. Class 4240 is a descendent of the root class 4245, and is designated class 4002 in the display. Class 4241 is also a descendent of the root class 4245, and is designated class 4003 in the display. In addition, class 4247 is a descendent of the root class 4245, and is designated class 4006 in the display. Class 4241 has two descendants: class 4246 and class 4243. Class 4246 is designated class 4004 in the display shown in Figure 216. And class 4243 is designated class 4005 in the display. In the illustrated example, the user selected class 4243. If the CSL is granted in the method 4230 depicted in Figure 213, the class 4243 becomes the selected class, becomes highlighted 4244, and associated attributes 4242 are displayed.

Figure 214 is a flow diagram that represents a process 4231 of opening a class to view subclasses. Referring to Figure 216, in this example, the user double clicks on the class 4241 to be opened, and a request for a CSL is issued in step 4236 of Figure 214. If the CSL is granted, the method proceeds to step 4237, and the display of the class 4241 changes from a closed folder (such as is shown in Figure 216 for class 4240) to an open folder 4241, and all subclasses 4246 and 4243 are displayed. The step 4237 of obtaining a CSL for the open class 4241 is illustrated in the method shown in Figure 214 as a single step, it being understood that step 4237 comprises multiple steps similar to steps 4233, 4234 and 4235 shown in Figure 213.

Figure 215 is a flow diagram representing the steps of a process that occurs when a user selects the "find class" activity in step 4238. (The find class activity is a class search through the class hierarchy or schema). A class matching a search

pattern is first selected using the process 4230 depicted in Figure 213. If the process 4230 depicted in Figure 213 is successful, then the class is opened using the process 4231 depicted in Figure 214. It will be understood by those skilled in the art that the steps 4230 and 4231 shown in Figure 215 correspond to multiple step procedures shown in Figures 213 and 214, respectively.

The lock manager 4125 maintains a lock table for each class in the schema, and for each lock holder. This may be better understood with reference to Figures 217-219.

Figure 217 is a diagram of a schema 4248 corresponding to the display of Figure 4216, and it illustrates corresponding internal lock states of the classes 4245, 4240, 4241, 4246, 4243, and 4247 in the schema 4248. Figure 218 illustrates a lock table 4250 maintained by the lock manager 4125 and corresponds to the schema 4248 depicted in Figure 217 and displayed in Figure 216. Figure 219 is a diagram that illustrates the contents of a lock object 4260 for class 4243 in the lock table 4250 shown in Figure 218.

The schema 4248 that is displayed in Figure 216 can be diagrammed as shown in Figure 217 to show the internal lock states of the classes 4245, 4240, 4241, 4246, 4243, and 4247 that are maintained by the lock manager 4125. The processes described in Figure 214 and Figure 215 for opening and selecting classes have been performed on the schema 4248 shown in Figure 217. Class 4245 and class 4241 have been opened. Class 4243 has been selected.

Lock states are kept in the lock table 4250 by the lock manager 4125. The rows identified by reference numerals 4251, 4252, 4253, 4254, and 4255 of the lock table 4250 each corresponds to a class 4245, 4240, 4241, 4246, 4243, and 4247, respectively, in the schema 4248. Each lock holder has a corresponding column, which are shown in Figure 218 as lock objects 4256, 4257, 4258, and 4259. The lock table elements correlate the class handles 4251, 4252, 4253, 4254, and 4255 of the classes 4245, 4240, 4241, 4246, 4243, and 4247 in the schema 4248 with the lock objects 4256, 4257, 4258, and 4259. Class handle 4251 in the lock table 4250 has a CSL lock object 4261 associated with lock holder 4257 because the class 4245 in the schema 4248 is open on the display 4116 of the user who is lock

holder 4257. The class 4241 in the schema 4248 has a CSL 4262 because the user who is lock holder 4257 also has it open. Class 4243 in the schema 4248 has a CSL lock object 4260 because it is the selected class. Of course, the lock object 4269 corresponding to the class handle 4254 for this lock holder 4257 is empty in Figure 218, because the corresponding class 4246 shown in Figure 217 has no locks. Similarly, the lock object 4249 is blank or empty in Figure 218, because the corresponding class 4240 shown in Figure 217 has no lock applied to it.

An example of an element 4260 of the lock table 4250 corresponding to the selected class 4243 is shown in Figure 219. The contents of the lock object 4260 for class 4243 include means for indicating whether any lock types are applied to the corresponding class 4243. In the illustrated embodiment, a CSL count 4263 indicates that one class share lock exists for this class 4243. A lock holder handle 4267 is used by the lock manager 4125 to identify each lock holder. When a request for a new lock holder 4226 is granted, (see Figure 212), a lock holder handle 4267 is assigned to the new lock holder. Thus, the procedure for granting a request for a new lock holder includes the step of assigning a lock holder handle to the new lock holder. In the illustrated example, each user has a unique user identifier or user ID. The lock object 4260 includes a record of the user ID 4268 of the user who corresponds to the lock holder handle 4267. Because a single user may be a multiple lock holder, the user ID 4268 for other lock holders 4256, 4258 or 4259 may be the same as the user ID 4268 for the lock object 4260.

In the example shown in Figure 219, the lock holder 4267 has a class lock on class 4243, but does not have any tree locks (TSL, TUL, or TXL) on the class 4243. Thus a count for TXL locks 4264 is zero. Similarly, a count for TUL locks 4265 and a count for TSL locks 4266 are both zero in this example.

Figure 220 diagrams the process that occurs when the user discussed above with reference to Figures 216-219 adds a part to a class 4243 in the knowledge base 4123. When the user selects the 'make part" function in step 4270 using the retriever 4130 to add an instance to the database 4123, the client 4131 requests a tree update lock in step 4271 for the selected class 4243. If

the request for a TUL is successful, the flow proceeds to step 4273 and the user is given access to add the part. The TUL is then released by the lock manager 4125 when the add part operation is completed. If the request for a TUL is not granted, the flow proceeds to step 4272, and the user is denied access for the purpose of making a part. In a preferred embodiment, the user is given a message when access is not permitted to inform him or her of such event in step 4272.

Figure 222 depicts the states of the lock table 4250 for the process of adding a part as described in Figure 220. Figure 221 diagrams the schema 4248 to which the part is being added. A part is being added to class 4243 in the schema 4248 shown in Figure 221. Figure 224 shows the screen display during the process of adding a part under these circumstances. In order to perform the function of adding a part, a tree update lock is required. If it is granted, the lock object 4260 for class 4243 will have a TUL for the add part operation, and will also have a CSL as shown in Figure 222, since the class 4243 in the schema 4248 is also the selected class. Of course, the lock object 4269 corresponding to the class handle 4254 for this lock holder 4257 is empty in Figure 222, because the corresponding class 4246 shown in Figure 221 has no locks. Similarly, the lock object 4249 is blank or empty in Figure 222, because the corresponding class 4240 shown in Figure 221 has no lock applied to it.

Figure 223 shows the lock object 4260 in this example. The count 4265 for the tree update lock type is one in this example, because a TUL has been granted to this lock holder for this class 4243. As noted above, the lock holder also has a CSL for the class 4243, and the count 4263 for the class share lock type is also one in this example. Like reference numerals in Figures 217-219 and Figures 221-223 refer to like elements, and the description in connection with Figures 217-219 will not be repeated.

When step 4273 in Figure 220 is performed in a preferred embodiment, the step of opening an "add part window" 4275 (shown in Figure 224) is also performed. The tree 4276 under the influence of the tree update lock is represented in the add part window 4275 by a diagram 4276 representing class 4245, class 4241, and class 4243.

Referring to Figure 225, when the user selects the edit parts function in step 4280, the system clones the existing class share locks in step 4281 for the corresponding portion of the hierarchy 4248 currently displayed as a result of navigation to that point using the retriever 4130. Referring to Figure 227, the edit parts function creates a new window 4283 which contains a view of the class tree 4285 corresponding to the schema 4248. To present that additional view 4283 of the class hierarchy tree 4285, new share locks must be obtained for the presented classes 4245, 4241, and 4243. This insures a consistent view 4285 for the parts that are going to be displayed or edited in this window 4283. The system will reissue identical navigation locks for the parts editor tree 4285.

While in the edit parts window 4283, the user may navigate (see step 4282 in Figure 226) through the schema 4248 shown in Figure 228 to different locations in the class hierarchy tree 4285. This navigation uses the same navigation steps 4230 and 4231 described above, as shown in Figure 226.

In Figure 229, the lock holder table 4250 for this user is shown after the completion of the creation of the edit parts window 4283. Note that the lock holder 4257 contains two CSL (class share locks) 4261, 4262, and 4260 for each class 4245, 4241 and 4243 which has been opened to get to the class 4243 identified by class handle 4255. There is one CSL for each class 4245, 4241 and 4243 opened for the original retriever window 4290, and one CSL for each class 4245, 4241 and 4243 opened for the edit parts window 4283. When the user continues to navigate down the tree 4285, CSL's will be obtained for each class through which the user navigates.

Figure 230 shows the lock object 4260 for this example in more detail. The class share lock count 4263 is two, because two class share locks are contained in the element 4260 shown in Figure 229 at the intersection of the row corresponding to the class handle 4255 and the column corresponding to the lock object 4257.

Like reference numerals in Figures 228-230 refer to like elements in Figures 217-219 and Figures 221-223. Therefore, the description in connection with Figures 217-219 and Figures 221-223 will not be repeated.

Figure 231 depicts a flow chart for the method used when a user is attempting to move a selected part 4330 from one class 4243 to another class 4241 within a given subtree 4248. Figure 232 shows a flow chart for the method used in the general case of moving any number of parts from one class 4243 in a subtree 4248 to another class 4241 within that subtree 4248. The difference between these two figures is determined by the number of parts to be moved. In the special case of one part to be moved, an optimization can be made in the method shown in Figure 231 that makes it more likely that the operation can be completed because the locks are applied to a smaller set of composite objects.

In Figure 232, the general case of moving any number of parts from one class 4243 in a subtree 4248 to another class 4241 within that subtree 4248 is shown. In a preferred embodiment, this method is used to move more than one part, and the method depicted in Figure 231 is used to move a single part. In Figure 232, the operation begins by attempting to get a TXL (tree exclusive lock) for the subtree 4243 that was selected when the edit parts operation was started (Figure 225). If the lock is denied, then the operation is rejected. If the lock is granted, then a TUL (tree update lock) is requested for the destination class 4241. If the TUL is granted, then the all of the necessary locks are held, and the parts are moved from the source class 4243 to the destination class 4241.

Figure 231 shows the special case where only one part 330 is to be moved. The only thing that is different here is where the TXL is requested. Although the previous case (Figure 232) is known to work, it will be less likely to succeed because it requires a broader lock on the subtree 4285 in which parts are being moved. To increase the likelihood of moving the part 4330, the TXL is applied to the class 4243 which owns the instance 4330 being moved. This applies the lock to the smallest possible portion of the tree 4285, thereby locking the fewest number of instances. When the lock is granted, the operation proceeds in the same manner as the general case shown in Figure 232.

Referring more specifically to Figure 231, the user initiates the process in step 4300. Although this step is labeled "user moves one part," it should be understood that the first step is

more accurately an attempt by the user to perform the indicated function (if the necessary locks are available). The concurrency control system then proceeds to step 4301, in which the system requests a TXL for the class 4243 that owns the part that is to be moved. Although Figure 231 refers to the "defining class of part," those skilled in the art will appreciate that it is more accurately referred to as the owning class for that part. If the TXL cannot be obtained, (because the lock manager 4125 detects a conflicting lock present in the lock table 4250), the system proceeds to step 4302. The GUI preferably informs the user that the requested move cannot be performed, for example, with a message that access is denied because the information is in use. If the requested TXL is granted by the lock manager 4125, the system proceeds to step 4303. The system requests a TUL on the destination class 4241. If the requested tree update lock cannot be granted, (because the lock manager 4125 detects a conflicting lock present in the lock table 4250), the system proceeds to step 4304 and preferably informs the user that the requested move cannot be performed. If the requested TUL does not conflict with an existing lock in the lock table 4250, the lock manager 4125 grants the requested TUL and proceeds to step 4305. The part can then be moved.

The dynamic class manager 4134 will, of course, perform operations on the objects in the knowledge base 4123 which are described more fully above.

Figure 233 shows the lock table 4250 for the general case of moving parts described in Figure 232. The table 4250 of lock holders may have multiple locks on many portions of the tree 4285. The illustrated lock table 4250 identifies the locks that are held by this lock holder 4257 (the locks held by the retriever 4290, the locks held by the parts editor 4283, and the locks held for the move parts operation). The classes 4245, 4241, and 4243 identified by class handles 4251, 4253, and 4255, respectively, each have a CSL (class share lock) for the retriever, and a CSL for the parts editor. In addition, the class 4241 identified by class handle 4253 has a TUL (tree update lock) to add the part that is about to be moved to the class 4241. Also, the class 4243 identified by class handle 4255 has a TXL for removing the part from the class 4243.

In the illustrated example, in order to have the TXL granted to the class 4243 represented by class handle 4255, there may be no other TSL (tree share locks), TUL, TXL, or CSL's held by other lock holders 4256, 4258 or 4259 which are currently operating. The fact that there are CSL's held by this lock holder 4257 is considered a self conflicting condition. This condition is permitted, and the TXL granted, due to the fact that the CSL locks are held by the requester 4257 of the TXL. In general, in circumstances like this, broader locks are granted if and only if the only conflicts that can be identified are with the lock holder 4257 that is making the request.

A preferred display for moving a selected part 4330 is shown in Figure 235. In the preferred embodiment, the display of the parts editor window 4283 shown in Figure 235 should visually indicate the source class 4243 in the tree 4285 with a dotted rectangle 4291, highlighting, color code, or some other distinguishing feature. The destination class 4241 should be visually indicated by highlighting 4292, or some other distinguishing feature. The user accomplishes the move function by clicking on the move command button 4335.

Figure 236 depicts a flow chart for the optimized case where one part 4328 is to be removed from the knowledge base 4123. The process is started at step 4320. In step 4321, a TXL is requested for the class 4243 that owns the instance 4328 that is to be removed. Although Figure 236 refers to the class 4243 as the "defining class of part," those skilled in the art will appreciate that the class 4243 is more accurately referred to as the owning class. If the TXL cannot be obtained, then the operation is denied in step 4322. If the operation succeeds, the TXL is granted in step 4323, and the part 4328 is deleted.

Figure 238 shows the locks 4260 that must be held by a lock holder 4257 that wishes to remove an instance 4328 from a class 4243. This condition is essentially the same as a portion of the move operation (see Figure 233) wherein the part must be removed from a class 4243. The lock conditions are the same for the classes 4245 and 4243 represented by class handles 4251 and 4255, respectively. The class 4241 represented by class handle 4253 holds a CSL for the retriever, and a CSL for the parts editor. To delete a part 4328, a TXL must be held for the class 4243 from

which the instance 4328 or set of instances is to be removed.

Figure 237 depicts a flow chart for the general case of deleting more than one part from the subtree 4248, which begins with step 4324. The TXL is requested in step 4325 from the class 4245 that was identified when the part editor was invoked. This is the class 4245 that defines the subtree 4285 wherein work will be done. After successfully obtaining the TXL, instances are deleted from their owning classes 4245, 4240, 4241, 4246, 4243, or 4247. If the TXL is denied, then the operation is rejected, and no parts are deleted.

Figure 239 shows the preferred display associated with the delete parts operation. A part 4330 is selected by clicking on the display 4330 of the part. The selected part 4330 corresponds to the part 4328 to be deleted shown schematically in Figure 238. In the illustrated example, the selected part is defined by, or a member of, the selected class 4243. The deletion of the selected part 4330 is initiated by clicking on the delete command button 4331 shown in Figure 239. In a preferred embodiment, if the requested locks are granted, the system opens a dialog box or window 4332 as shown in Figure 240 to ask the user to confirm that he or she wants to delete the selected part 4330. The deletion operation is confirmed by clicking on the "yes" button 4333, at which time the dialog box 4332 is closed and the selected part 4330 is deleted.

Figure 241 describes the steps that are involved in concurrency control when using the schema editor to change the structure of the schema. In step 4340 when the user selects the schema developer or schema editor 4144, the next operation is to obtain a TXL lock on the subtree that the user wishes to modify. The procedure for attempting to accomplished this begins with step 4341, where a tree exclusive lock is requested for the active class 4243. If the TXL cannot be obtained, then the process branches to step 4342 and the schema developer 4144 cannot be started. When the TXL lock is granted, the method proceeds to step 4343 and the schema developer screen 4350 is displayed. Following step 4343, the CSL locks that were obtained for the retriever 4290 on the class 4243 that was selected for schema editing are released in step 4344 (because a TXL lock has been obtained for that class 4243). In step 4345, a CSL lock is

then obtained by the schema developer 4144 for the parent class 4241 of the class 4243 in which the schema is to be edited. A CSL is preferably also obtained for the parent class 4245 of the class 4241.

Figure 242 shows a lock table 4250 that indicates the locks that are held during the operations described in Figure 241. Note that the schema editor 4144 holds a TXL (in element 4260) for the class 4243 represented by class handle 4255. This prevents other users of the system from accessing any of the information in the subtree below the class 4243 represented by class handle 4255. Details of the lock object 4260 are also shown in Figure 242.

Figure 243 illustrates a screen display for a preferred embodiment showing a schema developer window 4350 that is opened in step 4343 of Figure 241. The class 4243 in which the schema is to be edited is preferably displayed in a highlighted manner 4349.

Figure 244 shows a flow chart illustrating the mechanisms that are used by the concurrency control means when displaying a instance. The operation begins at step 4360 by the user selecting the parts display mechanism. Referring to Figure 246, this is initiated when the user clicks on the display command button 4352. In order for the system to display the desired information, there will need to be locks present. In order to obtain locks, the software 4131 must become a lock holder. A request for a new lock holder is performed in step 4361 shown in Figure 244. If the request to become a lock holder is denied, then the flow proceeds to step 4362 and the user is not allowed to display the parts. However, if the lock holder request is granted, then the flow proceeds to step 4363 and the software 4131 requests a TSL (tree share lock) on behalf of the user. If the TSL is denied, then the method proceeds to step 4362 and the operation cannot proceed. When the TSL is granted for the active class, the method proceeds to step 4364 and parts can be displayed with a confidence that the information contained within that subtree 4243 is coherent.

Figure 245 depicts the lock table 4250, a diagram of the schema 4248, and details concerning one of the lock objects 4373. Figure 245 shows the condition of the lock holder table 4250 for

the situation described in Figure 244. The retriever 4290 is holding the CSL locks 4261, 4262 and 4260 for all of the classes 4245, 4241 and 4243, respectively, that have been navigated through to get to the class 4243 that is represented by class handle 4255. In order for the system to display the parts defined by class 4243, a new lock holder 4258 is formed, and a TSL lock 4373 is requested for the class 4243 identified by class handle 4255. The TSL 4373 insures that other locks cannot be granted, and hence there will be no ability to modify the schema 4248, or the instances contained within the this subtree 4243. Hence the list of parts 4365 displayed in the search results window 4351 shown in Figure 246 will be consistent, and coherent for the duration that the lock 4373 is held.

In Figure 245, the tree share lock is indicated in the lock table 4250 at the intersection 4373 of the row corresponding to the class handle 4255 and the column corresponding to the lock holder 4258. The lock object 4373 is shown in more detail in Figure 245. The TSL count for the lock object 4373 is shown as one, because the lock holder 4258 holds a TSL lock for the class 4243 corresponding to the class handle 4255. The lock holder handle 4372 is two, indicating that this is a different lock holder from the lock holder described above with reference to Figures 238 and 242. However, the user ID 4370 is shown as 4100. This is because the same user (whose user ID is 4100) is two lock holders.

Figure 253 is a chart representing the application of the lock manager 4125 by the knowledge base client 4131. When a retriever window 4502 is opened, the concurrency system assigns the user the lock holder 4001 (LH1) position, and grants CSL's to that lock holder. To perform a "find class" function 4503, lock holder 4001 will require CSL's as shown in Figure 253. When a part edit window 4504 is opened, lock holder 4001 must obtain a TSL on the current class, and CSL's to navigate the hierarchy. Tree locks (TUL and TXL) are required to edit the schema.

In order to open a part display window 4505, the user will have to be assigned a new lock holder (LH2), and will require a TSL. In order to open a schema edit window 4506, Figure 253 shows that LH1 will require a TXL. In order to open an add part, or make part, window 4507, LH1 will require a TUL. In a

preferred system, a user can tear off a window 4508 in the retriever. In order to do so, the user must be assigned LH3, and will require CSL's to navigate the schema.

The present invention may include a knowledge base client means and a knowledge base server means. The knowledge base server means preferably comprises an object oriented lock manager means. The knowledge base server means preferably includes a dynamic class manager means, a connection manager means, a query manager means, a handle manager means, a units manager means, a database manager means, and a file manager means.

Figure 258 shows the major components of a computer hardware configuration 4109 providing the computational and communications environment for a knowledge base server 4132. This configuration consists of a central processing unit or CPU 6109 which includes an arithmetic logical unit 6100 which fetches and executes program instructions from main memory 6101. The programs are stored on a disk drive 4110, access to which is provided through a disk controller 6106. The knowledge base files 4123 are also stored on disk drive 4110 and accessed through virtual memory addresses 6112 in main memory 6101, through which, when required, a page 6111 of contiguous data in a disk file 6108 is copied into main memory 6101. The preferred embodiment of the present invention uses virtual memory 6112 for this knowledge base management system. The knowledge base server 4132 interacts with the client API 4143 through a local area network 4100, access to which is controlled by network controller 6102, or through a wide area network 6104, access to which is controlled by a serial interface controller 6103. An I/O bus 6105 mediates data transfers between the CPU 6109 and the peripheral data storage, interface and communication components.

Figure 259 shows the major components of a computer hardware configuration 4112 providing the computational and communications environment for a retriever 4130, schema editor 4144, a graphical user interface component of legacy 4133, and an API 4143. This configuration consists of a central processing unit or CPU 6109 which includes an arithmetic logical unit 6100 which fetches and executes program instructions from main memory 2601. The programs are stored on one or more disk drives 6110, access to which is provided through a disk controller 6106. The user interacts with

the system through the keyboard 4115 and mouse or similar graphical pointer 4114 with the graphical user interface displayed on the CRT display 4113. The API 4143 communicates with the knowledge base server 4132 through a local area network 4100, access to which is controlled by network controller 6102, or through a wide area network 6104, access to which is controlled by a serial interface controller 6103. An I/O bus 6105 mediates data transfers between the CPU 6109 and the peripheral data storage, interface and communication components.

The present invention may be advantageously used in a client/server architecture comprising a knowledge base client and a knowledge base server, as shown in Figure 204. However, the invention is not necessarily limited to a client/server architecture. The invention may also be used in a distributed database system.

C. Object Oriented Database Structure

Figure 267 depicts a flow chart showing the procedure followed when a user edits parts. Referring, for example, to **Figure 266**, a user who has access rights to edit parts may actuate the edit button 7180 and bring up the parts editor window 5019 shown in **Figure 292**. The first step 5012 shown in **Figure 267** involves the user selection of attributes and parts to edit from the parts editor window 5019. A user may enter new or revised values 5061 for attributes 5101, and the system will accept parameter input in step 5013. If the attribute is an enumerated attribute 5101, a pull down list 5062 will be presented to the user with available choices, as shown in **Figure 293**. In step 5014 of **Figure 267**, a determination is made as to whether there are more parts to edit. If there are no more parts to edit, flow proceeds to step 5017. The system updates the part display 5020 and the parts editor window 5019 with edited values 5061. The system then proceeds to step 5018 and returns control to the user.

In step 5014, if more parts remain to be edited, flow proceeds to step 5015, and the system gets the next selected part. In step 5016, the system sets the next selected parts parameter to the user input value 5061. Control then loops back to step 5014.

Figure 294 depicts a procedure for deleting parts. In step 5021, the user selects parts to delete from the edit parts window 5019. The user then clicks a delete parts command button 5026.

In step 5022, a determination is made as to whether any more parts remain to be deleted. If the answer is yes, flow proceeds to step 5023 in which the system gets the next selected part and deletes it from the query result and the knowledge base. Flow then loops back to step 5022. When there are no more parts to delete, flow proceeds to step 5024, and the system redisplays the updated query result in the part editor window 5019. Flow then proceeds to step 5025, and control is returned to the user.

Figure 295 depicts a flow chart for a procedure for moving parts. The procedure may be initiated by the user selecting parts to move from the parts editor window 5019 as shown in step 5102. Alternatively, the user may initiate the procedure as in step 5103 by navigating the class hierarchy on the parts editor window 5019 and selecting a destination class. The user may actuate a move parts command button 5027, which is illustrated for example in **Figure 284**.

Referring to **Figure 295**, the procedure proceeds to step 5104 and a determination is made as to whether there are more parts to move. If there are no more parts to move, flow transfers to step 5042 and the system redisplays the query result in the parts editor window 5019. The flow then proceeds to step 5043, and control is returned to the user.

Returning to step 5104 in **Figure 295**, if a determination is made that there are more parts to move, flow proceeds to step 5105 and the system gets the next selected part. In step 5106 a determination is made as to whether the user has requested an unconditional move. If the answer is yes, flow jumps to step 5040. The system then sets the part class to the destination class selected by the user. Any parameters or missing attributes are set to undefined. Flow proceeds to step 5041, and the system deletes the moved part from the query results. Flow proceeds to step 5042 where the system redisplays the query result in the parts editor window 5019.

In step 5106, if the user has not requested an unconditional move, flow proceeds to step 5107 where a determination is made as to whether attributes for any part parameters are missing from the destination class. If the answer is no, flow proceeds to step 5040 and continues as described above.

If a determination is made in step 5107 that there are

attributes for part parameters which are missing from the destination class, flow transfers to step 5108. The system gets a list of parameters that will be deleted by the move and presents them to the user by displaying them on the display 4116. Flow then proceeds to step 5109. If the user then overrides the warning that parameters will be deleted, or requests that the parts be moved unconditionally, flow transfers to step 5040 and proceeds as described above. If the user does not wish to override the parameter deletion warning or does not request that the parts be moved unconditionally, flow loops back to step 5104.

The process of editing parts may be further understood in connection with a description of the parts editor window 5019 (shown in **Figure 284**). Once the user has specified a part by selecting a class 7174 and subclasses 7196, 7197, 7198 and 7199, entered the attribute search criteria 7177, and set the display order 4194, the user can edit the parts by choosing the edit command button 7180. Choosing this command 7180 causes the parts editor window 5019 to appear. The top area 5102 of the parts editor window 5019 contains the class tree 5044, showing highlighting the navigation path and class definition of the parts the user is editing. The bottom area 5103 of the window 5019 contains the parts 5020 the user has selected to edit. The parts appear in a table 5020 that is similar to tables that are used in spreadsheet applications. The part attributes 5049, 5100, 5101, etc., and attribute values 5105, 5106, 5107, etc., appear in the display order, from left to right, that the user previously established in the part specification window 7170. To use a value, the user clicks an enter box 5063. To cancel a new value, the user clicks a cancel box 5064.

The top area 5102 of the parts editor window 5019 contains the class definition 5044, which comprises the class tree showing the navigation path and class definition of the parts selected for editing. The window 5019 has a horizontal split bar 1047 that splits the window into two sections 5102 and 5103. The user can move the split bar 5047 up or down so the user can see more of one section 5102 or the other 5103. The parts editor window 5019 includes an area referred to as the editing area 5046. After selecting an attribute value 5101, a text box or list box 5104 appears in this editing area 5046 so the user can make changes

(see **Figure 292**). Each part appears as a row 5048 in the table 5020,and each row 5048 of the table 5020 is numbered. The user may use the row number to select a part that the user needs information on or that the user wants to move or delete. The attributes 5049, 5100, 5101, etc., are the column headings, and the attribute values are the rows.

After determining that the user is going to enter a new part in the knowledge base, the user must fully specify the part. In a preferred embodiment, a complete part specification is defined as selecting the class up to the leaf class 7201 and entering values for all the required attributes 7203. In a preferred embodiment, if the user does not select a leaf class 7201 or enter the required attributes 7203, the user cannot add the part. When making parts, a preferred procedure is for the user to enter as many attribute values 7203 as the user can in order to give the part as complete a specification as possible.

Some attributes are required before a part can be added. Before choosing the make command 7181, the user must enter an attribute value for each required attribute. In addition, a user cannot enter any attribute values for protected attributes. Protected attributes have a protected icon 7191 immediately to the left of the attribute icon. Once the user has selected the leaf class 7201 and entered all required attributes, the user can choose the make command button 7181. Choosing the make command 7181 causes the part to be added to the user's knowledge base and the parts found 7172 to be updated to show a part count of 4001.

The knowledge base client 4131 is a set of C++ libraries that provide knowledge base services to a client application 4130, 4133, and 4144 through the API 4143. The services may be either local or result in remote procedure calls to the knowledge base server 4132. For client applications which run under Windows, the knowledge base client consists of one or more Windows Dynamic Link Libraries (DLL) which use the WinSock DLL to provide network access to the knowledge base server 4132 and the registry server 4141.

The knowledge base server 4132 is a UNIX server process that manages knowledge base 4110 access, retrieval and updates. A knowledge base server 4132 may manage one or more knowledge bases 4110 and 4110.

The dynamic class manager 4134 is a software subsystem in the knowledge base server 4132 that manages schema and data. The dynamic class manager 4134 provides the ability to store class, attribute, unit and instance information that can be modified dynamically. The dynamic class manager 4134 consists of C++ libraries and classes and provides operations for "legacizing" and for accessing, creating, deleting, and modifying classes, attributes, instances, parameters, unit families, units and meta-attributes at run-time.

The capabilities of the dynamic class manager 4134 are accessed by a user programmer through a set of functions provided by the API 4143.

The dynamic class manager 4134 knowledge base, hereafter sometimes referred to as "the knowledge base," is a collection of classes, attributes, units, instances with parameter values, and relationships among these objects. In the dynamic class manager 4134, a class defines a separate type of object. Classes have defined attributes. The attributes have some type, and serve to define the characteristics of an object. A class can be derived from another class. In this case, the class inherits attributes from its ancestors. A knowledge base contains instances of classes. The attribute values defined by an instance are parameters.

Another way to describe the concept of classes, attributes, instances, and parameters is to use a dog as an example. The word "dog" is the analog of a class. Dog describes a group of similar things that have a set of characteristics, or attributes. The attributes of a dog are things like color, breed, and name. The class and attributes do not describe any particular dog, but provide the facility to describe one. An instance of a dog has parameters that give values to the attributes: for example, a dog whose color is beige, of the breed golden retriever, and whose name is Sandy.

Classes can have relationships. The class "dog" is part of the larger class, "mammal". The class "mammal" is less specific than "dog". It conveys less information about the object "dog", but everything about "mammal" also applies to "dog". "Dog" is clearly a subset of "mammal", and this relationship is a subclass. "Dog" is a subclass of the class "mammal". The

subclass "dog" could be further subclassed into classes like big "dogs", little "dogs", etc. The concept subclass implies a parent relationship between the two classes. "Mammal" is a parent and "dog" is a subclass. The terminology "'dog' is derived from 'mammal'" is also used to describe the relationship.

The subclass "dog" inherits attributes from its parent class. The attribute color could be part of the "mammal" class, since all "mammals" have a color. The "dog" class inherits the attribute color from its parent.

The root class is special, it has no parent. It is the class from which all classes begin their derivation. In illustrations set forth herein, graphs have been drawn to illustrate a class hierarchy, and the root class is placed at the top of those drawings. Subclasses branch out from the root class into ever widening paths that make the graph look like an upside down tree. The entire group of classes is a tree, and the special class that has no parent, though it is at the top, is the root.

One of the available attribute types supported by the dynamic class manager 134 is a numeric type. Numeric attributes are used to describe measurable quantities in the real world. Such measurements do not consist of just a numeric value; they also have some associated units. The dynamic class manager 4134, in conjunction with the units manager 4138, maintains information about different types of units that can be used with numeric attributes. The dynamic class manager 4134 (using the units manager 4138) can also perform conversions among units where such conversion makes sense. The units that the system understands are grouped into various unit families. These unit families and the units they define, can be changed at run time. The dynamic class manager 4134 also comprises a dynamic units manager 4138.

The word "schema" refers to the layout of classes, attributes, units, and unit families. A knowledge base with no instances is a schema. This may be better understood in connection with the following more detailed description of the various objects managed by the dynamic class manager 4134.

A class is the most fundamental object in the schema in accordance with the present invention. A class is a collection of related objects. In the present example, a class may have eight or nine components. A class is a schema object. As

explained above, the schema is the collection of classes, attributes, units, and unit families and their relationships. Every class has exactly one parent from which it is derived, except for the root class 7173. The root class 7173 is the one class that has no parent. The root class 7173 has another special characteristic in that it can never be deleted. The consequence of a class being derived from its parent means that the class has all of the properties of its parent. These properties are referred to as attributes. Attributes are inherited from the parent class.

A class may have zero or more subclasses. A class is a parent of each of its subclasses. A subclass is a class that has a parent, so the root class 7173 is not a subclass. The subclasses of a parent class have some defined order. The order is persistent, meaning that the dynamic class manager 4134 preserves the order even across closes and reopens of the knowledge base.

A class has a set of descendants that is comprised of all of its subclasses, all of the subclasses' subclasses, and so on. A class that has zero subclasses or an empty set of descendants is called a leaf class 7201. A subtree is the set composed of a class and all of its descendants. The subtree is said to be rooted at the class. A subclass also has a set of ancestors, which is the set composed of the parent, its parent's parent, and so on including the root class 7173. Classes that have the same parent are sometimes referred to as siblings.

Following a subclass to its parent is sometimes referred to as going up the tree. Moving from a parent to one of its subclasses is sometimes referred to as going down the tree. Therefore, the root class 7173 of the schema is the furthest up at the top of the tree, and the objects furthest down at the bottom of the tree are typically leaf classes 7201.

A class has a name which is the text identifying a class, subclass, or leaf class, and is an ASCII character string. The present invention uses class handles for references to a class, which are further described in connection with the operation of the handle manager 7137. In the example shown in **Figure 264**, there are three subclasses.

Figure 285 shows the internal object representation for a class 4800. In the present schema, a class has a parent handle

4801. Every class object 4800 includes stored information representing the handle of its parent class, except in the special case of the root class 7173, which has no parent. A null is stored in this location in that case. A handle is a reference to an object. The parent handle information 4801 is used by the handle manager 7137 to identify the stored class object which is the parent class for the class 4800.

The class object 4800 includes a subclass list 4802. The subclass list 4802 is an array of handles which may be used by the handle manager 7137 to identify those class objects which are subclasses of the class 4800. In the internal representation provided in the present invention, lists can grow without bounds and are dynamic. The storage space available is not fixed.

This provides flexibility and power to the database structure, because the class object 4800 may have an extremely large number of subclasses in a large database without substantial degradation in performance.

The class object 4800 includes an attribute list 4803. The attribute list 4803 is a list of handles. The handle manager 7137 may use the information stored in the attribute list 4110 to identify the attributes possessed by class object 4800.

The class object 4800 also includes a local instance list 4804, which is a handle list. Field 4805 shown in **Figure 285** is a pointer to storage location of the class name, i.e., the text identifying the class.

Field 4806 is used to store the handle for the class 4800. The field 4807 stores an indication of the class code, i.e., whether it is primary, secondary, or a collection.

The class object 4800 also includes a subtree instance count 4808. The subtree instance count 4808 is a numeric indication of the total number of items or instances present in all of the descendants of the class 4800 i.e., the total number of instances in class 4800, all of the class 4800's subclasses, all of the subclasses' subclasses, and so on. Thus, when a user is navigating through the tree structure of a knowledge base, as a user selects and opens subclasses, the user can be immediately informed of the number of parts found at any location on the tree by retrieving the subtree instance count 4808 for the current class and passing that information to the retriever 4130. The

subtree instance count 4808 is kept up to date whenever the knowledge base is modified, so it is not necessary while a user is navigating through the tree structure of the database to perform a real time computation of parts found 7172.

Referring again to **Figure 285**, the class object 4800 also preferably includes a metaparameter list 4809. The metaparameter list 4809 is a string list, and may be used as a pointer to strings containing linking information ,for example, the name of a file that contains a graphical display of the type of parts represented by the class 4800, thesaurus information used for legacizing data, or other legacizing information.

Figure 286 depicts an example of a generic list 4810. The class manager 4134 uses lists of handles, lists of floating point values, lists of pointers to character strings, etc. whenever a variable amount of data can be associated with an object. Examples of lists would be items 4802, 4803, 4804 and 4809. The list 4810 depicts a list of simple integers.

A list object 4810 includes a pointer 4812 which points to the beginning 4815 of the list data 4811. A list object 4810 also includes a field 4813 indicating the currently allocated size for the list data 4811. The list object 4810 also includes a field 4814 containing information indicating the amount of list data 4811 currently in use.

The list data 4811 contains the actual list of values. The first item 4815 in the list in this example contains the value "4005". Similarly, in this example list items 4816, 4817, 4819, 4820 and 4821 contain additional values. List items 4822, 4823, 4824, 4825 and 4826 in this example are not currently in use and are set to zero. In this illustrated example, the currently allocated size 4813 of the list is twelve. The amount in use 4814 of the list is seven, meaning that the first seven items in the list are valid.

Figure 287 illustrates the data structure for attribute data 4827. An attribute object 4827 contains at least six fields in the illustrated embodiment. A first field 4828 contains a pointer to an external name comprising an ASCII character string that is the name for the attribute. The attribute object 4827 also contains a field 4829 containing the handle for this attribute object 4827. The attribute object 4827 also contains

a field 4830 which contains the handle of the class that defines this attribute 4827. The fourth field 4831 is a Boolean indication of whether this attribute is a required attribute for the defining class. A fifth field 4832 contains a Boolean field indicating whether this attribute is protected. This is indicated by the protected icon 7191. In the data structure of the attribute object 4827 shown in **Figure 287**, this information is stored in field 4832. The attribute object 4827 also contains a field 4833 which is a metaparameter list.

Enumerated attributes include fields 4828 - 4833, indicated collectively as attribute data 4834, plus a field 4835 which is a list of enumerator handles.

In the case of a Boolean attribute, only fields 4828 - 4833 are used, which are again indicated collectively in **Figure 287** as attribute data 4834.

Numeric attributes include fields 4828 - 4833, indicated collectively as attribute data 4834, plus a field 4838 which contains the handle of the unit family for this numeric attribute.

In the case of a string attribute, and in the case of a string array attribute, only the attribute data 4834 comprising fields 4828 - 4833 is included.

One example of the use of these data structures by the dynamic class manager 4134 is the procedure of a user selecting a class by clicking on the closed folder icon 7189 associated with the class. When a class is opened, the dynamic class manager 4134 will check the class object 4800 and retrieve the attribute list 4803. The handles stored in the attribute list 4803 will be passed to the handle manager 7137. The handle manager 7137 will return the virtual memory address for each attribute 4827 of the class. The dynamic class manager 4134 may then use the pointer 4828 to the external name of an attribute object 4827 to retrieve the character string text for the external name for the attribute. That ASCII text information can then be passed through the API 4143 so that it may eventually be provided to the retriever 4130 for display to a user on the display 4116.

Figure 288 illustrates the data structure for an enumerator object 4841. An enumerator object 4841 may comprise three fields. A first field 4842 contains a pointer to the external

name for the enumerator object 4841. A second field 4843 contains the handle for the enumerator object 4841. A third field 4844 may contain a metaparameter list. Handles are used to link from other objects to the enumerator object 4841. An advantage of this structure is the ability to easily modify a knowledge base if it becomes desirable to change the external name of an object. Such a change need only be performed once to the ASCII character string that is used to represent the external name. All other objects merely contain a handle which can be used by the handle manager 7137 to provide the dynamic class manager 4134 with the actual external name.

Figure 289 depicts the data structure for an instance 4871 and associated parameters 4872. An instance object 4871 may contain four fields 4873 - 4876. The first field 4873 is the handle for the owner class of this instance. The second field 4874 may give the ordinal location of this instance's handle in the instance list 4804 of its owning class. The third field 4875 is a list of parameters, which points to the values contained in 4877. The fourth field 4876 is the handle for the instance object 4871. The list of parameters 4877 contains a plurality of pointers to parameters for the various attributes associated with this instance object 871. In the example illustrated in **Figure 289**, the list 4877 contains three entries 4878, 4879 and 4880. Additional elements of the list 4877 have been omitted for clarity. The pointer 4878 in list 4877 points to information concerning the associated parameter 4872. The data structure for the parameter 4872 is illustrated in more detail in **Figure 290**.

Figure 290 shows the data structure for five different types of parameters: enumerated, Boolean, numeric, string and string array. Each of the parameter objects 4872 has an attribute handle 4881. An enumerated object 4888 has an attribute handle 4881 and an enumerator handle 4882. A Boolean object 4889 has an attribute handle 4881 and a Boolean value 4883. A numeric parameter object 4890 has an attribute handle 4881, a unit handle 4884 and a value 4885. For example, if the numeric parameter is 4010 ohms, the unit handle 4884 would be the handle for the ohms unit, and the value 4885 would be 4010. A string parameter 4891 contains a field for the attribute handle 4881 and a pointer 4886 to an ASCII character string. A string array parameter 4892

contains an attribute handle 4881 and a field 4887 that points to a list of pointers to string arrays.

Figure 291 is an example of a schema with instances. The example has a class named "electronics", which has a subclass 4800' named "capacitors". The capacitors subclass 4800' has an attribute 4827 called "case type". There are two possible types of cases in this example, which are referred to as "case A" and "case B". The subclass capacitors 4800' has a subclass 4800' named "electrolytic". The electrolytic subclass 4800' has an attribute 4827' called "voltage rating", and one instance 4871' is provided that has parameters 4890 and 4888 of 5 volts and a type B case, respectively. Most objects and lists are shown incomplete in order to simplify the illustration, it being understood that like reference numerals refer to the same objects described in connection with **Figures 273-278**.

In **Figure 291**, the class object 4800 has a name 4806, which in this case is "electronics". The class object 4800 has a field 4802 which points to a list of subclasses 4893. The list 4893 has a first entry 4894 which is the handle for the subclass 4800'. In this case, the name 4806' of the subclass 4800' is capacitors. Of course, all references to schema objects actually use handles (not shown in **Figure 291**) and actually go through the handle manager 7137 and handle table. This is not shown in **Figure 291** in order to simplify the diagram.

The subclass 4800' capacitor has a field 4802' which points to a list of subclasses 4893'. The list 4893' has an entry 4894' which is the handle for subclass 4800". The name 4806" for subclass 4800" is electrolytic. The subclass 4800" has a null entry in the field 4802" which would normally contain a pointer to a list of subclasses, if any. In this example, the subclass 4800" does not have any subclasses.

Returning to the capacitors subclass 4800', field 4803 contains a pointer to a list of attributes 4897. The list 4897 contains the handle for the enumerated attribute 4827 called "case type". Field 4830 of the enumerated attribute object 4827 contains the handle of the defining class 4800' called capacitors. The enumerated attribute object 4827 contains a pointer 4835 which points to a list 4839 of handles for enumerators. In this example, the list 4839 contains a handle

4898 for the enumerator 4841. The enumerator 4841 contains a pointer 4842 to the external name for this enumerator, which may be an ASCII string for "case A". Similarly, item 4899 in the list 4839 points to enumerator 4841' associated with case B.

Returning now to subclass 4800" named electrolytic, the pointer 4803" points to a list 4897' of attributes, and one of the fields in the list 4897' contains the handle for numeric attribute 4827' which is "voltage rating". The numeric attribute 4827' contains a field 4830' which contains the handle of the defining class which in this example is the class 4800" named electrolytic. The numeric attribute object 4827' also contains a field 4838' which contains the handle of the voltage unit family (not shown).

Returning to the electrolytic class 4800", a field 4804" contains a pointer to a list 4895 of handles of instances. Item 4896 in the list 4895 contains the handle associated with instance 4871. Instance 4871 contains a field 4873 which contains the handle of the owning class, which in this case is the electrolytic class 4800". The instance data object 4871 also contains a field 4875 which points to a list of parameters 4877. The list 4877 contains a pointer 4878 which points to the numeric parameter 4890. The numeric parameter 4890 contains a field 4881 which contains the handle of the attribute 4827' (voltage rating). The numeric parameter object 4890 also contains a field 4884 which has the handle of the units, which in this case is "volts". For simplicity, the unit object is not shown. The numeric parameter object 4890 contains a field 4885 which contains the value 5.0. In this instance, the electrolytic capacitor is rated at 5.0 volts.

The parameter list 4877 contains a pointer 4879 which points to the enumerated parameter 4888. The enumerated parameter object 4888 contains a field 4881' which contains the handle of the attribute, which in this instance is case type. The enumerated parameter object 4888 also contains a field 4882 which is the handle for the enumerator 4841'. In this example, the electrolytic capacitor rated at 5.0 volts has a type case B.

The data structure described herein has significant advantages. Referring to **Figure 291**, it is easy to change a name or description in this data structure. Consider an example where

the database may contain 1,000 instances of capacitors with a type B case. If the type B case is discontinued, or the name changed to "re-enforced", the only change that would need to be made would be to replace a single ASCII string representing the name for that case type. All 1,000 instances in the database simply contain a handle that the handle manager 7137 associates with that ASCII text string. No other changes need to be made in the database.

Another advantage of the data structure in accordance with the present invention is that if a primary value is undefined, nothing is stored. Thus there is no wasted space.

Another advantage of the database structure is that algorithms do not have to be changed based upon location in the tree structure. All algorithms work the same regardless of location in the tree structure. The only special case is the root class. For example, the algorithm for adding an instance to the database is the same no matter where in the tree structure you are located. This makes dynamic changes to the schema very easy. A class or an entire branch of the tree structure can be moved from one location to another simply by changing lists of handles. It is not necessary to run a convert program. Everything is self contained. A class object 4800 contains the handle of its parent 4801 and thus knows who it's parent is. The class object 4800 also contains a pointer 4802 to a list of its subclasses, so it knows who its children are.

In the present database structure, it is possible to delete instances quickly. An instance can be deleted by taking the last item in the list of instances 4804 and moving it to the position of the instance being deleted. In other words, the handle of the last instance would be written over the handle of the instance being deleted, and the number of items in the list would be decremented by one. The instance index field 4874 for an instance object 4871 may be used to facilitate fast deletions.

In a preferred embodiment, the value of parameters are always stored in base units. The objects in fields described do not necessarily occupy a word of memory. In a preferred embodiment, all parameters of a particular type are stored contiguously. This improves the speed of searches. For example, the case type 4841' described with reference to **Figure 291** would be stored

contiguously with all the other parameters for case type. The numeric parameter of 5.0 volts would be stored in a different physical location in memory contiguous with other numeric volt parameters.

As described above, providing a class object structure 4800 with a field 4808 providing the subtree instance count for that class allows the system to virtually instantly display a parts count 7172 to provide the user instantaneous feedback during the tree traversal steps of the users search. The process of finding a part essentially amounts to discarding the thousands of parts that do not have the attributes desired and narrowing the search down to a small number that do.

This is accomplished by navigating to the correct class from the root of the classification hierarchy. During this phase, the parts found indication 7172 can be updated using the data structure field 808 indicating the subtree instance count. This provides significant response time advantages compared to actually counting the available instances at each step. The user has immediate feedback indicating the number of parts available in the selected tree. The combination of providing an object oriented hierarchical tree structure together with search criteria based upon any desired combination of attributes, while providing instantaneous feedback on the number of instances corresponding to the current search criteria and class provides significant advantages over data base management schemes that have been attempted in the past.

An important function of the dynamic class manager 4134 is the ability to modify the database structure during operation. The database structure is known as the schema. The schema of the object oriented database is structured using classes. The classes contain attributes. The attributes may contain enumerators, and unit families. The ability to add, move and delete these items is important to the dynamic operation of the database.

To add a class to the schema, three items must be known: the class name, the parent of the new class, and the location within the list of subclasses to insert the new class. **Figure 292** illustrates this operation. The first step 5840 converts the handle of the parent class into an actual class pointer. The

parent pointer must be immediately tested in step 5841 prior to its use. If the pointer proves to be invalid, then the operation terminates at step 5842. If the pointer is valid, the insertion index is tested in step 5843. If it proves to be invalid, the operation is terminated in step 5844. Finally, the name of the class must be tested in step 5845 to determine if it fits the guidelines of valid class names. If the class name fails, then the operation terminates in step 5846. When step 5845 accepts the class name, the new class can be created. A new handle is created in step 5847 first, and then the new class is created in internal memory in step 5848. The new handle is inserted into the table of class handles in step 5849 of **Figure 293**, followed by the handle being added to the parents list of subclass handles in step 5850. The last operation is to cause the file manager 4140 to add the new class to the indicated parent on the secondary storage device 4110.

To add an attribute to a class, three items must be known: the class handle of the owning class, the location in which to insert the new attribute, and the name of the attribute. **Figure 294** illustrates the adding of attributes. The first step 5930 is to convert the class handle into a class pointer, followed by the testing of that class pointer in 5931 to determine if it is a valid class pointer. If not, the procedure terminates in 5932. If the class pointer is determined to be valid, then the insertion index is validated in 5933. If the index fails the validation test, then the procedure terminates in 5934. If the validation of the index succeeds, then the operation continues in 5935 where the name of the attribute is tested. If the attribute name fails, then the operation terminates in 5936. If the name of an enumerated attribute is accepted in 5935, then the attribute can be created. Step 5937 creates a new handle for the attribute. Then the new attribute is created in step 5938. The new attribute handle is then added to the list of attributes local to the owning class in 5939. The last step is 5940 of **Figure 295** to cause the file manager 4140 to update secondary storage 4110 with the new attribute. The operation is complete in step 5941.

The addition of an instance is shown in **Figure 284**. Adding an instance requires a class handle. The class handle must be

converted into a class pointer in 5918. The class pointer is tested in 5919 to determine if it is a valid class pointer. If it is not valid, then the procedure terminates in 5920. If the class pointer is determined to be valid, then the procedure continues in 5921 with the generation of a new instance handle and a new instance object. The handle for the new instance is inserted into the handle table in 5922. The instance is added to the parents list of instances in 5923. The subtree instance count is incremented to reflect the presence of the new instance in 5924. The instance has now been created in memory, and needs to be added to secondary storage 4110, which is done in step 5925 of **Figure 285**. The procedure is complete in step 5926.

The deletion of a class is shown in **Figure 286**. To remove a class from the database structure, the current class handle must be identified. The class handle is first decoded into a class pointer in step 6600. The class pointer is then checked to determine if it is a valid class pointer in 6601. If the class pointer is invalid, the procedure is terminated in 6602. If the class pointer is valid, then it is checked to determine if it is the root class in 6603. If the class pointer represents the root class, then the procedure terminates in 6604, because the root class cannot be deleted. If the class pointer does not represent the root class, it is checked to determine if the class represents a leaf class in 6605. If the class pointer does not represent a leaf class, the procedure terminates in 6604. If the class pointer is found to point to a leaf class, then operation continues in 6906 where all of the instances of this class are deleted. The process of deleting instances is described below with reference to **Figure 290**. In step 6607 all of the attributes which are local to the class being deleted are deleted. In **Figure 287** The class is then unlinked from its parent class in step 6608. The system checks to determine if the unlink was successful, and that the data structures which contain the class list are intact in 6609. If the unlink failed, then operation stops in 6610. If the unlink succeeded, then operation continues in 6611 where the class object is actually deleted. In step 6612, the file manager 4140 is instructed to remove the class object from secondary storage 4110, and the operation completes in step 6613.

The deletion of an attribute is shown in **Figure 288**. To remove an attribute, the attribute handle must be decoded into an attribute pointer in step 5860. Step 5861 checks to see if the attribute pointer obtained from step 5860 is valid. If the attribute pointer is invalid, the procedure stops in 5862. If the attribute pointer is valid, the procedure continues in step 5863 by searching the entire subtree for all of the parameters in all of the subtree instances that are derived from this attribute. After searching, in step 5864 the system determines how many parameters were derived from this attribute. If there were parameters derived from this attribute, the operation proceeds to 5865, where the parameters are undefined. If there were no parameters derived from this attribute, then the procedure continues to step 5866. Likewise, after the parameters have been undefined in 1865, the operation continues to 5866. In step 5866, the attribute is unlinked from the defining class. In 5867 the procedure checks to determine if the unlink operation succeeded. If the unlink failed, then the procedure stops at 5868. If the unlink was successful, then the attribute object is deleted in 5869 in **Figure 289**. The file manager 4140 is then instructed to remove the attribute from secondary storage 4110 in step 5870. The operation is complete in step 5871.

The deletion of an instance is shown in **Figure 290**. An instance is deleted from the database by first converting the instance handle in step 6000 to an instance pointer. The instance pointer is checked to determine that it is indeed a valid instance pointer in 6001. If the instance pointer is invalid then the operation terminates in 6002. If the instance pointer is valid, then the instance is unlinked from its owning class in 6003. The instance object is itself deleted in 6004. The subtree instance counts is then decremented to indicate that one instance has been deleted from the subtree in 6005. The file manager 4140 is then instructed to update the secondary storage 4110 to reflect the deletion of the instance in 6006. The operation is complete in step 6007.

In **Figure 291**, moving a subtree to a new position in the class hierarchy is described. In step 5800, the move subtree procedure is called with a class to move, the destination parent class, and the position among its sibling classes at the destination

specified. In step 5801, the class pointers for the class to be moved and the destination parent class are obtained. If the test for all valid pointers in step 5802 fails, step 5804 returns an error, else test 5805 is made to prevent the class from being trivially moved to its own parent. Step 5806 insures that the position among the subclasses of the destination parent class is within a valid range, with an error returned by step 5804 upon error. In step 5807, the class hierarchy above both the class to be moved and the destination class is analyzed to identify the nearest common ancestor class.

In step 5808 of **Figure 292**, the common ancestor is tested to see if it is identical to the class being moved. If it is, given that a test has already been performed to insure that the class is not being moved to its parent, then this is determined to be an attempt to move a class to a subclass of itself, and an error is returned. All other moves are legal, so the class is unhooked from its parent class in step 5809 and added to the list of subclasses for the destination class in step 5810. The destination class subtree instance count is incremented by the number of instances in the moved class in step 5811, and the subtree count of the original parent class of the moved class is decremented by the moved class instance count in step 5812. In step 5813 the permanent image of the knowledge base is updated through the file manager 4140, with step 5814 returning successfully to the caller.

Figure 293 describes unhooking the moved class from its original parent class. In step 5815 the class pointer for the parent is obtained and used in step 5816 to get a list of subclasses for the parent class. If the class handle of the class to be moved is not in the resulting subclass list as tested in step 5817, the knowledge base is internally inconsistent and an error is returned to the caller, else the class is deleted from the parent class subclass list in step 5818 before a successful return in step 5819.

Figure 294 describes the process of finding the nearest common ancestor of the class to be moved and the destination class. In step 5820, a temporary class handle is set to the handle of the class to be moved. Step 5821 gets the parent of the temporary class, initiating a loop that creates a list of classes in order

from the class to move to the root. Each class encountered is added to a list in step 5822, with iteration being terminated if step 5823 shows that the root has been encountered. If the test in step 5823 fails, the temporary class handle is set to the handle of its parent class in step 5824 and iteration continues.

A similar list is created for the destination class in steps 5831 through 5828, moving to **Figure 295**. In step 5831, a temporary class handle is set to the handle of the destination class. Step 5832 gets the parent of the temporary class, initiating a loop that creates a list of classes in order from the class to move to the root. Each class encountered is added to a list in step 5826, with iteration being terminated if step 5827 shows that the root has been encountered. If the test in step 5827 fails, the temporary class handle is set to the handle of its parent class in step 5828 and iteration continues.

The final step 5829 iterates through the two resulting lists until a matching class handle is found. This is the handle of the nearest common ancestor, which is returned in step 5830.

D. Comparing Instances By Their Attribute Values

A preferred method and apparatus for performing a search or query is described in more detail in application Serial No. 08/339,481, filed Nov. 10, 1994. When the results of a search are obtained, the instances (in the illustrated example the instances are parts) may be displayed as shown in Figure 262A. The parts may then be compared by their attribute values. The parts that the user wishes to compare are selected by clicking on them. When selected, the display of the selected part is shaded or highlighted, as shown in Figure 262A by the shaded part displays indicated by reference numerals 4653 - 4662. The parts 4663, 4664, and subsequent parts are not highlighted because they have not been selected in the illustrated example.

After displaying the list of parts that match the search specification (see Figure 262A), a user will often want to compare those parts in relation to their shared attribute values. This can be done by using the compare parts option 4652 from the actions menu 4651. This command 4652 accesses the part attribute comparison dialog box 8630 shown in Figure 262B and Figure 263, where a user can compare the attribute values among all selected parts 4633, 4634, 4635, and 4636. In a preferred embodiment, a

user must select at least two parts in the search results window 4650 before invoking the compare parts command 4652. An example of comparing a selected part's attribute values to all other values is shown in the part attribute comparison dialog box 4630 shown in Figure 262B.

In a preferred embodiment, before the part attribute comparison dialog box 4630 first appears on the user's display as shown in Figure 262B, all attribute values for the selected parts are evaluated as to whether or not they have the same value. When the part attribute comparison dialog box 4630 appears on the user's display screen, the results of this comparison are indicated in the first column 4637 of the dialog box 4630. In the illustrated example, an equal operator (=) 632 is displayed in the first column 4637 where all of the attributes in that row 4643 are equal for all of the parts 4633, 4634, 4635, and 4636 selected for the compare parts operation. A not equal (<>) operator 4631 is displayed in the first column 4637 where all of the attributes in that row 4642 are not equal for all of the parts selected for the compare parts operation. The second column 4638 of the dialog lists the attribute titles, and the remaining columns 4633, 4634, 4635, and 4636 are each allocated to a single part; that is, one for each part that the user previously selected from the search results window. Each part column 4633, 4634, 4635, and 4636 lists its attribute values in the same order as the other columns.

An example part attribute comparison dialog box 4630 is shown in Figure 262B. **Table 13** describes the regions of the part attribute comparison dialog box 4630:

Table 13

Region	Description
Initial Evaluation 4637	Displays either an equal operator (=) or a not equal (<>) operator, when the part attribute comparison dialog box 630 first appears. An equal operator 4632 indicates that all the values for that specific attribute are the same for all the selected parts. A not equal operator 4631 indicates that at least one value, for all the same attributes, for all the selected parts, is not the same.
Attribute Title 4638	Displays the name of each attribute in a separate row.
Part 4633, 4634, 4635 and 4636	Displays the values for each attribute for a particular part. Each attribute value is an element in a separate row for the column corresponding to that part. A part (column) number is at the top of the column corresponding to each part.

Referring to Figure 262B and Figure 263, certain command buttons 4639, 4640 and 4641 are provided in the illustrated embodiment. A "compare to selected part" command button 4639 causes the system to compares all the attribute values of the other parts 4633, 4634, and 4636 shown in the dialog box 4630 to those belonging to a single part 4635 a user has selected (see Figure 263). The user must select the part 4635 by clicking on its column number 4635 (labeled "part 4003" in Figure 4060) before choosing this command 4639. A "clear comparisons" command button 4640 causes the system, once a comparison has been conducted using the "compare to selected part" command 4639, clears the comparison results (at which point the display will return to a display similar to that shown in Figure 262B). A

"close" command button 4641 will cause the system to close the part attribute comparison dialog box or window 4630 and return to the display window that was active before the compare parts dialog box 4630 was opened. **Table 14** describes the command buttons 4639, 4640 and 4641 in the part attribute comparison dialog box 4630.

Table 14

Command	Description
Compare to Selected Part 4639	Compares all the attribute values shown in the dialog box to those belonging to a single part a user has selected. The user must select the part (that is, its column number) before choosing this command.
Clear Comparisons 4640	Once a comparison has been conducted using the Compare to Selected Part command, clears the comparison results.
Close 4641	Closes the dialog box.

Referring to Figure 263, when the compare to selected part command is issued, the attribute display changes to indicate the results of the comparison in a way that makes equal and unequal comparisons immediately apparent to a user in a very convenient manner. When all of the attribute values for the non-selected parts 4633, 4634, and 4636 are compared to those for the baseline part 4635, the cells 4644 and 4645 for attribute values that match are not shaded, and the cells 4647 for attribute values that do not match are shaded. For example, the selected or baseline part 4635 has a value for the attribute "major material" 4648 indicating that the part is made of "steel" 4646. The attribute value "steel" 4646 for the selected part 4635 is compared to the values of the other parts for the attribute "major material" 4648. The first part 4633 has a value of "steel" 4644 for this attribute 4648. Because it is the same value 4644 as the attribute value 4646 for the selected part

4635, it is displayed unshaded, as shown in Figure 263. Similarly, the second part 4634 also has a value of "steel" 4645 for this attribute 4648. Because it is the same value 4645 as the attribute value 4646 for the selected part 4635, it is also displayed unshaded, as shown in Figure 263. The fourth part 636 has a value of "nylon" 4647 for this attribute 4648. Because it is not the same as or equal to the attribute value 4646 for the selected part 4635, it is displayed as a shaded cell 4647, as shown in Figure 263.

A procedure for comparing part attributes may include the following steps:

1 From the search results window, a user selects two or more parts that the user wants to compare.

2. From the actions menu, the user chooses compare parts. The part attribute comparison dialog box 4630 appears, showing which attribute values for a single attribute are the same (=) 4632, or if any attribute values for a single attribute are different (<>) 4631, in the first column 4637.

3 Referring to Figure 263, to compare the attribute values for all parts 4633, 4634, and 4636 displayed in the dialog box 4630 with those for a baseline part 4635, the user clicks the baseline part column number 4635 and chooses the compare to selected part command button 4639. All the attribute values for the non-selected parts 4633, 4634, and 4636 are compared to those for the baseline part 4635. The cells 4644 and 4645 for attribute values that match are not shaded; the cells 4647 for attribute values that do not match are shaded.

4 To clear the color comparisons, the user chooses the clear comparisons command button 4640.

5 To compare the attribute values for all parts displayed in the dialog to those for a different part, repeat step 4003. Figure 260 and Figure 261 depict flow charts for the process of comparing part attributes. In step 4625, the user selects a number of parts greater than one for comparison. Of course, the user must select more than one part, because there would be nothing to compare with the baseline part if only one part was selected. In step

4626, the user invokes a compare parts command 4652 from an action menu 4651.

In step 4627, a window or dialog box is opened and the parts selected for comparison are displayed. In Figure 262A, the part attributes are preferably displayed in rows. In a preferred embodiment, the part attributes are preferably displayed in columns as shown in Figure 262B. The user then selects in step 4628 a part 4635 to compare. Point A identified with reference numeral 4629 is a common point in the flow charts of Figures 260 and 261.

Step 4630 is an entry point into an outer program loop, and step 4631 is an entry point into an inner program loop. In step 4632, the system checks to determine whether the current instance is the selected baseline instance 4635. If it is, the method jumps to step 4635 and goes to the next instance or column. If it is not, the method proceeds to step 4633 where the method determines whether the corresponding attribute values are the same (or match) for the current instance and the selected instance 4635. If the attribute values are equal, the display of that cell 4644 of the attribute row 4648 is unchanged, and the flow proceeds to step 4635, where the procedure goes to the next instance. If the attribute values are not equal, the method goes to step 4634, and the display of that cell 4647 of the attribute row 4648 is changed, for example to highlight it, or the background color is changed, or it is shaded. The flow then proceeds to step 4635 and goes to the next instance, or column. In step 4636, a check is made to determine whether this is the last instance for this attribute, i.e., whether it is the last column. If not, the process loops to step 4631. If it is the last instance for this attribute, i.e., it is the last column, the procedure goes on to conduct a comparison of the next attribute, i.e., it goes to the next row. In step 4637, the method checks to see if this is the last row. If not, the process loops back to step 4630. If it is the last row, the comparison has been completed for all rows and columns, i.e., each attribute has been compared for every instance. The system then exits at step 4638.

E Summary

Using the present invention, in a preferred application

involving the use of an object oriented database management system to manage parts information, multiple users may access the same knowledge base 4123 concurrently for finding parts, editing parts, and editing the schema. The object oriented database management system manages concurrency by using "locks."

More than one schema editor or developer 4144 can be active concurrently in the same knowledge base. When a user selects the class that he or she wants to edit, the schema editor 4144 establishes a lock on that class. As long as the schema editor 4144 has a lock on that class, that class and all of its subclasses are not accessible for editing in any other schema editor 144, and may not be available for viewing by the retriever 4130. However, another schema editor 4144 and/or retriever 4130 may concurrently work on any other section of the knowledge base 4123 that does not have a lock.

With schema developer/retriever concurrency, a user can edit his or her schema 4123 at the same time that the rest of his or her company is using the object oriented database management system to retrieve parts information. Anyone attempting to find or edit parts in the area that is locked, preferably receives a message indicating that the class is locked. When this message appears, the first user can either go to a different area of the knowledge base 4123 or wait until the second user's schema editor 4144 releases the lock.

All of the editing functions require an application to become a lock holder and then request a form of write lock before the edit will succeed.

F. Software Functions

The enumeration, ***pmx_lockType***, is used to specify the lock types that can be requested and released for classes in the knowledge base.

```
typedef enum {
  PMX_ERROR_LOCKTYPE          = 0,
  PMX_NO_LOCK                 = 1,
  PMX_CLASS_S_LOCK            = 2,
  PMX_TREE_S_LOCK             = 3,
  PMX_TREE_U_LOCK             = 4,
  PMX_TREE_X_LOCK             = 5
} pmx_lockType;
```

The enumeration, **pmx_lockMode**, is used to describe the lock state of a class in the knowledge base. Any given class is in some lock state which is defined by the locks present on the class, either explicitly on the class or by virtue of the class being in a subtree which is locked.

```
typedef enum {
  PMX_LOCKMODE_ERROR            = 0,
  PMX_LOCKMODE_NONE             = 1,
  PMX_LOCKMODE_SHARE            = 2,
  PMX_LOCKMODE_UPDATE           = 3,
  PMX_LOCKMODE_EXCLUSIVE        = 4,
} pmx_lockMode                  = 5
```

The **pmx_lockDescriptor** structure returned by the API function pmx_getLockDescriptor to return information about the locks held by the specified lock holder at the specified class. The specified class and lock holder are returned along with number of times each type of lock has been acquired.

```
typedef struct {
  pmx_classHandle               classHandle;
  pmx_lockHolderHandle          lockHolderHandle;
  long                          classShareLockCount;
  long                          treeShareLockCount;
  long                          treeUpdateLockCount;
  long                          treeExclusiveLockCount;
} pmx_lockDescriptor;
```

The following API functions are preferably provided for concurrency control:

pmx_startLockHolder
pmx_endLockHolder

pmx_requestLock
pmx_releaseLock
pmx_releaseAllLocks
pmx_releaseAllLocksOfType

pmx_freeLockDescriptor
pmx_getLockDescriptor
pmx_getLockMode

pmx_equalLockHolderHandles
pmx_isNullLockHolderHandle
pmx_getNullLockHolderHandle

These functions are used to start and end being a lock holder, to request and release locks, and to retrieve information about the lock status of classes.
Lock holders, which are identified by lock holder handles, are started and ended with *pmx_startLockHolder()* and *pmx_endLockHolder()*.
To request a lock, use pmx_requestLock. To release a lock or group of locks, use *pmx_releaseLock()*, *pmx_releaseAllLocks()*, or *pmx_releaseAllLocksOfType()*.
To retrieve information about the locks that have been acquired on classes, use *pmx_getLockMode()* or *pmx_getLockDescriptor()*.

A description of these functions follows:

whichDB	The handle of an open knowledge base.
lockHolder	The handle of a lock holder which has been started.
thisClass	The class of the class for which lock information is desired.

Description

This function returns the count of the locks of each type which have been acquired for the given lock holder and class. Only the locks which have been requested for the given class are reported.
A class may be influenced by a tree lock on an ancestor, but that condition is not reported.

The application should free the descriptor when it is finished with it. The application should also take care not to alter or destroy any of the data in the descriptor since the *pmx_freeLockDescriptor()* function expects it to be

uncorrupted.

Return Value

1. pmx_endLockHolder

Purpose

Terminate a lock holder that has been started.

Syntax

```
cd_boolean
pmx_endLockHolder(
    pmx_dbHandle           whichDB,
    pmx_lockHolderHandle   lockHolder );
```

Parameters

whichDB	The handle of an open knowledge base.
lockHolder	The handle of a lock holder which has been started.

Description

The lock holder is ended. Any locks that were requested with the lock holder handle are automatically released. The function will fail if the lock holder handle is invalid (i.e., it has never been started).

Return Value

Upon success, returns CD_TRUE.

Upon failure, returns CD_FALSE.

Errors

PMX_ERRORBADDBHANDLE
The knowledge base handle is invalid.

PMX_ERRORBADLOCKHOLDERHANDLE
The lock holder handle is invalid.

2. pmx_getLockDescriptor

Purpose

Get the description of the locks held at the given class.

Syntax

```
pmx_lockDescriptor CD_FAR *
pmx_getLockDescriptor(
    pmx_dbHandle          whichDB,
    pmx_lockHolderHandle  lockHolder,
    pmx_classHandle       thisClass );
```

Parameters Upon success, returns a pointer to the descriptor.

Upon failure, returns a NULL pointer.

Errors

PMX_ERRORBADCLASSHANDLE
The class handle is invalid.

PMX_ERRORBADDBHANDLE
The knowledge base handle is invalid.

PMX_ERRORBADLOCKHOLDERHANDLE
The lock holder handle is invalid.

3. pmx_getLockMode

Purpose

Return the lock mode of a given class.

Syntax

```
pmx_lockMode
pmx_getLockMode(
    pmx_dbHandle            whichDB,
    pmx_lockHolderHandle    lockHolder,
    pmx_classHandle         thisClass,
    cd_boolean              self );
```

Parameters

whichDB	The handle of an open knowledge base.
lockHolder	The handle of a lock holder which has been started.
thisClass	The handle of the class for which the lock mode is desired.
self	Specifies whether the lock mode is desired with respect to the current application (self) or all other applications.

Description

This function returns the lock mode of a given class. The lock mode is the effective lock on the class caused by locks at the class and at ancestors of the class. The application has the choice of asking for the mode based on locks it has acquired or based on locks held by other applications. When the self argument is CD_TRUE, then the lock mode result is based on the locks acquired by the current application and lock holder. Otherwise, when self is CD_FALSE, then lock mode result is based on all other applications and lock holders.

Return Value

Upon success, returns the lock mode.

Upon failure, returns PMX_LOCKMODE_ERROR.

Errors

PMX_ERRORBADBOOLEANVALUE
A boolean value is not CD_TRUE or CD_FALSE.

PMX_ERRORBADCLASSHANDLE
The class handle is invalid.

PMX_ERRORBADDBHANDLE
The knowledge base handle is invalid.

PMX_ERRORBADLOCKHOLDERHANDLE
The lock holder handle is invalid.

4. pmx_releaseAllLocks

Purpose
Releases all the locks that have been acquired for a class and for all of its descendants.

Syntax

```
cd_boolean
pmx_releaseAllLocks(
    pmx_dbHandle          whichDB,
    pmx_lockHolderHandle  lockHolder,
  pmx_classHandle       thisClass );
```

Parameters

whichDB	The handle of an open knowledge base.
lockHolder	The handle of a lock holder which has been started.
thisClass	The handle of the class at the root of the subtree for which the locks are to be released.

Description
This function releases all the locks of all types held on all classes in the subtree rooted by the given class. Only the locks for the given lock holder are released. An error does not occur if no locks have been acquired.

Return Value
Upon success, returns CD_TRUE.

Upon failure, returns CD_FALSE.

Errors

PMX_ERRORBADCLASSHANDLE
The class handle is invalid.

PMX_ERRORBADDBHANDLE
The knowledge base handle is invalid.

PMX_ERRORBADLOCKHOLDERHANDLE
The lock holder handle is invalid.

5. pmx_releaseAllLocksOfType

Purpose

Releases all the locks of a given type held on all classes in the subtree rooted by the given class.

Syntax

```
cd_boolean
pmx_releaseAllLocksOfType(
    pmx_dbHandle          whichDB,
    pmx_lockHolderHandle  lockHolder,
  pmx_classHandle       thisClass,
  pmx_lockType          lockType );
```

Parameters

whichDB	The handle of an open knowledge base.
lockHolder	The handle of a lock holder which has been started.
thisClass	The handle of the class at the root of the subtree for which the locks are to be released.
lockType	The type of lock which is to be released.

Description

This function releases all the locks of the specified type held on all classes in the subtree rooted by the given class. Only the locks for the given lock holder are released. An error does

not
occur if no locks have been acquired.

Return Value
Upon success, returns CD_TRUE.

Upon failure, returns CD_FALSE.

Errors
PMX_ERRORBADCLASSHANDLE
The class handle is invalid.

PMX_ERRORBADDBHANDLE
The knowledge base handle is invalid.

PMX_ERRORBADLOCKHOLDERHANDLE
The lock holder handle is invalid.

PMX_ERRORBADLOCKTYPE
The lock type is invalid.

6. pmx_releaseLock

Purpose
Releases the lock of the given type that has been acquired on the
given class.

Syntax

```
cd_boolean
pmx_releaseLock(
    pmx_dbHandle          whichDB,
    pmx_lockHolderHandle  lockHolder,
  pmx_classHandle       thisClass,
  pmx_lockType          lockType );
```

Parameters

whichDB	The handle of an open knowledge base.
lockHolder	The handle of a lock holder which has been started.
thisClass	The handle of the class for which the lock is to be released.

lockType The type of lock which is to be released.

Description

This function releases one lock of the given type for the given class and lock holder. An application can acquire multiple locks of the same type for a single class, so the lock must be released as many times as it is requested. Locks can be released en masse with pmx_releaseAllLocks.

The function fails if the lock described by the lock holder, class handle, and lock type has not been previously acquired.

Return Value

Upon success, returns CD_TRUE.

Upon failure, returns CD_TRUE.

Errors

PMX_ERRORBADCLASSHANDLE
The class handle is invalid.

PMX_ERRORBADDBHANDLE
The knowledge base handle is invalid.

PMX_ERRORBADLOCKHOLDERHANDLE
The lock holder handle is invalid.

PMX_ERRORBADLOCKTYPE
The lock type is invalid.

PMX_ERRORNOSUCHLOCK
Attempt to release a lock which is not present.

7. pmx_requestLock

Purpose

Request that a lock of the given type be acquired on the given class.

Syntax

```
cd_boolean
pmx_requestLock(
    pmx_dbHandle           whichDB,
    pmx_lockHolderHandle   lockHolder,
    pmx_classHandle        thisClass,
    pmx_lockType           lockType );
```

Parameters

whichDB	The handle of an open knowledge base.
lockHolder	The handle of a lock holder which has been started.
thisClass	The handle of the class for which a lock is requested.
lockType	The type of lock which is requested.

Description

This function requests a lock of the given type for the given class and lock holder. The lock is acquired if the request does not conflict with the locks held by other applications and lock holders.

Return Value

Upon success, returns CD_TRUE.

Upon failure, returns CD_FALSE.

Errors

PMX_ERRORBADCLASSHANDLE
The class handle is invalid.

PMX_ERRORBADDBHANDLE
The knowledge base handle is invalid.

PMX_ERRORBADLOCKHOLDERHANDLE
The lock holder handle is invalid.

PMX_ERRORBADLOCKTYPE
The lock type is invalid.

PMX_ERRORCANNOTGRANTLOCK
The requested lock cannot be granted.

8. pmx_startLockHolder

Purpose

Start being a new lock holder.

Syntax
pmx_lockHolderHandle
pmx_startLockHolder(
pmx_dbHandle whichDB);

Parameters
whichDB The handle of an open knowledge base.

Description
This function creates a new lock holder, identified by a lock holder handle. The new lock holder may be used to request locks.
Locks from one lock holder conflict with another lock holder, even
for the same application.

Return Value
Upon success, returns a new lock holder handle.

Upon failure, returns pmx_NullLockHolder, a NULL lock holder handle.

Errors
PMX_ERRORBADDBHANDLE
The knowledge base handle is invalid.

---

9. pmx_freeLockDescriptor

Purpose
Free a pmx_lockDescriptor.

Syntax
cd_boolean
pmx_freeLockDescriptor(
pmx_lockDescriptor * thisDescriptor);

Parameters
thisDescriptor The lock descriptor to be freed.

Description
This function frees the memory associated with a lock descriptor.
No further reference to the descriptor may be made after calling

this function.

Return Value
Upon success, returns CD_TRUE.
Upon failure, returns CD_FALSE.

Error
PMX_ERRORNULLPOINTER
A NULL pointer was passed in place of a required input argument.

10. pmx_equalLockHolderHandles

Purpose
Compare two lock holder handles for equality.

Syntax
```
cd_boolean
pmx_equalLockHolderHandles(
    pmx_lockHolderHandle    handle1,
    pmx_lockHolderHandle    handle2 );
```

Return Value
If the two handles are equal, returns CD_TRUE.
If not equal, returns CD_FALSE.

11. pmx_getNullLockHolderHandle

Purpose
Get a NULL lock holder handle.

Syntax
```
pmx_lockHolderHandle
pmx_getNullLockHolderHandle();
```

Return Value
Returns a NULL lock holder handle.

12. pmx_isNullLockHolderHandle

Purpose
Check if a lock holder handle is the NULL handle.

Syntax
```
cd_boolean
pmx_isNullLockHolderHandle(
```

```
        pmx_lockHolderHandle    handle );
```

Return Value

If the specified handle is the NULL handle, returns CD_TRUE. Upon failure, returns CD_FALSE.

The above description is intended to be only an example of the invention, setting forth a presently preferred embodiment. Modifications and alternative embodiments will be apparent to those skilled in the art after having the benefit of this disclosure. The scope of the invention should not be limited to the particular example described herein. Instead, the scope of the invention is intended to be defined by the claims.

CLAIMS

WHAT IS CLAIMED IS:

1. A database management system, comprising:

an object oriented representation of information describing the characteristics of existing instances organized in a parent-child/class-subclass structure, wherein the internal representation of an instance is dependent upon information that is locally available from a class to which that instance belongs plus inherited attributes from a parent class;

means for querying said object oriented representation in a guided and iterative manner;

means for displaying search results; and,

means for selecting particular information pertaining to the characteristics which are to be displayed by the means for displaying.

2. The database management system according to claim 1, wherein:

at least one class in said object oriented representation of information is represented as a class object having a handle, said class object having a parent handle identifying the parent class of said class object, said class object having a subclass list, said subclass list comprising an array of class handles identifying the subclasses of said class object, said class object including an attribute list comprising a list of handles which may be used to identify attributes of said class object, said class object including a subtree instance count, said subtree instance count comprising a numeric indication of the total number of instances that belong to said class object and that are present in all descendants of said class object.

3. The database management system according to claim 2, wherein:

at least one class in said object oriented representation of information is represented as a second class object having a handle, said second class object having a parent handle identifying the parent class of said second class object, said second class object having a local instance list, said local instance list comprising an array of handles which can be used to identify the instances that belong to said second class object, said second class object including an attribute list comprising a list of handles which may be used to identify attributes of said second class object, said second class object including a subtree instance count, said subtree instance count comprising a numeric indication of the total number of instances that belong to said second class object and that are present in all descendants of said class object.

4. The database management system according to claim 3, wherein:

said means for displaying search results includes means for displaying a numeric value corresponding to said subtree instance count providing a numeric indication of the total number of instances that are present in a class that a user is navigating in a search and all descendants of said class.

5. The database management system according to claim 1, wherein:

said instances are represented as an owning class and a list of information with no additional storage allocated for undefined characteristics.

6. The database management system according to claim 3, further comprising:

legacy means to facilitate organization of existing data into a hierarchical, object-oriented schema having an object oriented representation of information describing the characteristics of existing instances organized in a parent-child/class-subclass structure.

7. The database management system according to claim 6, wherein:

said legacy means is operative to standardize descriptions of instances in content and format as a function of the type of instance and having variable field length descriptions that are not subject to arbitrary predetermined field length limitations.

8. The database management system according to claim 6, wherein:

said legacy means includes a rule system for unit measure conversion, providing that units of first user specified families of parts are automatically converted to a predetermined unit of measure, and providing that units of second user specified families of parts are not automatically converted to another unit of measure.

9. The database management system according to claim 6, wherein:

said legacy means includes means for transforming existing textual information about an instance into parametric values for said instance within a schema, and means for automatically estimating the class to which said instance should placed in a schema.

10. The database management system according to claim 3, further comprising:

means for performing parametric attribute searches on a hierarchical, object-oriented schema.

11. The database management system according to claim 3, further comprising:

means for searching said object oriented representation of information to retrieve all instances that exactly correspond to a predetermined set of search criteria, as well as instances that closely match said predetermined set of search criteria.

12. The database management system according to claim 3, further comprising:

means for searching said object oriented representation of information to retrieve all instances that exactly correspond to a predetermined set of search criteria, as well as instances that correspond to a subset of said predetermined set of search criteria.

13. The database management system according to claim 3, further comprising:

means for searching said object oriented representation of information to retrieve all instances that exactly correspond to a predetermined set of search criteria, as well as instances that correspond to a subset of said predetermined set of search criteria.

14. A network having a client/server architecture, comprising:

a knowledge base server, the knowledge base server including a dynamic class manager, a connection manager, a query manager, a handle manager, a units manager, a database manager, and a file manager;

an object oriented hierarchical schema representing classes of instances as objects arranged in a hierarchy, said schema being fully connected with each class object including information as to any class object that is a parent in the hierarchy, and class objects that are descendants in the hierarchy, said class objects being managed by said dynamic class manager; and,

an application programming interface to permit a client application to access the object oriented hierarchical schema.

15. The network according to claim 14, wherein:

at least one class in said object oriented hierarchical schema is represented as a class object having a handle, said class object having a parent handle identifying the parent class of said class object, said class object having a subclass list, said subclass list comprising an array of class handles identifying the subclasses of said class object, said class object including an attribute list comprising a list of handles which may be used to identify attributes of said class object, said class object including a subtree instance count, said subtree instance count comprising a numeric indication of the total number of instances that belong to said class object and that are present in all descendants of said class object.

16. The network according to claim 15, wherein:
at least one class in said object oriented representation of information is represented as a second class object having a handle, said second class object having a parent handle identifying the parent class of said second class object, said second class object having a local instance list, said local instance list comprising an array of handles which can be used to identify the instances that belong to said second class object, said second class object including an attribute list comprising a list of handles which may be used to identify attributes of said second class object, said second class object including a subtree instance count, said subtree instance count comprising a numeric indication of the total number of instances that belong to said second class object and that are present in all descendants of said class object.

17. The network according to claim 16, further comprising:
an object oriented lock manager, said object oriented lock manager being operable to allow modification of a first portion of said object oriented hierarchical schema by one client application while a plurality of other client applications are navigating or searching a second portion of said object oriented hierarchical schema.

18. A parts management system, comprising:

a processor;

a display having a screen, the display being coupled to the processor;

a mouse coupled to the processor;

a knowledge base accessible by the processor, the knowledge base having descriptive information for a plurality of parts corresponding to products of an organization, the knowledge base comprising a hierarchical schema of parts information representing classes of instances, the hierarchical schema of parts information having a root class, the hierarchical schema of parts information having a plurality of levels of descendant classes, the root class being a parent of a plurality of first level descendent classes, at least some of the first level descendant classes being first level parent classes of respective second level descendent classes, at least some of the second level descendant classes being second level parent classes of respective third level descendent classes, the classes and instances having a plurality of attributes, wherein a class at a level "n" in the hierarchical schema of parts information inherits attributes from its parent class at a level "n-1" in the hierarchical schema;

means for displaying a graphical tree hierarchy in a tree display area of the screen, the graphical tree hierarchy representing classes in a currently selected portion of the hierarchical schema of parts information, said classes having individually associated icons for individual classes displayed in the tree display area of the screen;

means for navigating the graphical tree hierarchy by clicking with the mouse on selected locations in the tree display area representing a portion of the hierarchical schema of parts information; and,

means for displaying attributes in an attribute display area of the screen, the attribute display area being distinct from the tree display area, the means for

displaying attributes being coordinated with the means for displaying classes such that the attributes displayed in the attribute display area are the corresponding attributes for a currently selected location in the hierarchical schema of parts information.

19. The parts management system according to claim 18, wherein:
at least one class in said object oriented hierarchical schema is represented as a class object having a handle, said class object having a parent handle identifying the parent class of said class object, said class object having a subclass list, said subclass list comprising an array of class handles identifying the subclasses of said class object, said class object including an attribute list comprising a list of handles which may be used to identify attributes of said class object, said class object including a subtree instance count, said subtree instance count comprising a numeric indication of the total number of instances that belong to said class object and that are present in all descendants of said class object.

20. The parts management system according to claim 19, wherein:
at least one class in said object oriented representation of information is represented as a second class object having a handle, said second class object having a parent handle identifying the parent class of said second class object, said second class object having a local instance list, said local instance list comprising an array of handles which can be used to identify the instances that belong to said second class object, said second class object including an attribute list comprising a list of handles which may be used to identify attributes of said second class object, said second class object including a subtree instance count, said subtree instance count comprising a numeric indication of the total number of instances that belong to said second class object and that are present in all descendants of said class object.

21. The parts management system according to claim 20, further comprising:

means for displaying a numeric value in a parts found display area corresponding to a subtree instance count representing the total number of instances that belong to a selected class and that are present in all descendants of said class when navigating the graphical tree hierarchy by clicking with the mouse on selected locations in the tree display area.

22. An object oriented database management system in a client/server architecture, comprising:

a knowledge base client;

a knowledge base server, the knowledge base server including a dynamic class manager, a connection manager, a query manager, a handle manager, a units manager, a database manager, and a file manager;

an object oriented hierarchical database structure including classes, where each class is represented by a class object data structure which includes hierarchical location identifying information for said class, said class object data structures being managed by said dynamic class manager;

an object oriented lock manager for controlling access by a plurality of client applications, said object oriented lock manager providing concurrency control using class share locks, tree update locks, and tree exclusive locks, and not using instance locks; and,

a lock holder table, the lock holder table being used by the lock manager to control concurrent access by said client applications by granting appropriate locks to a client application when the requested lock does not conflict with an existing lock in the lock holder table.

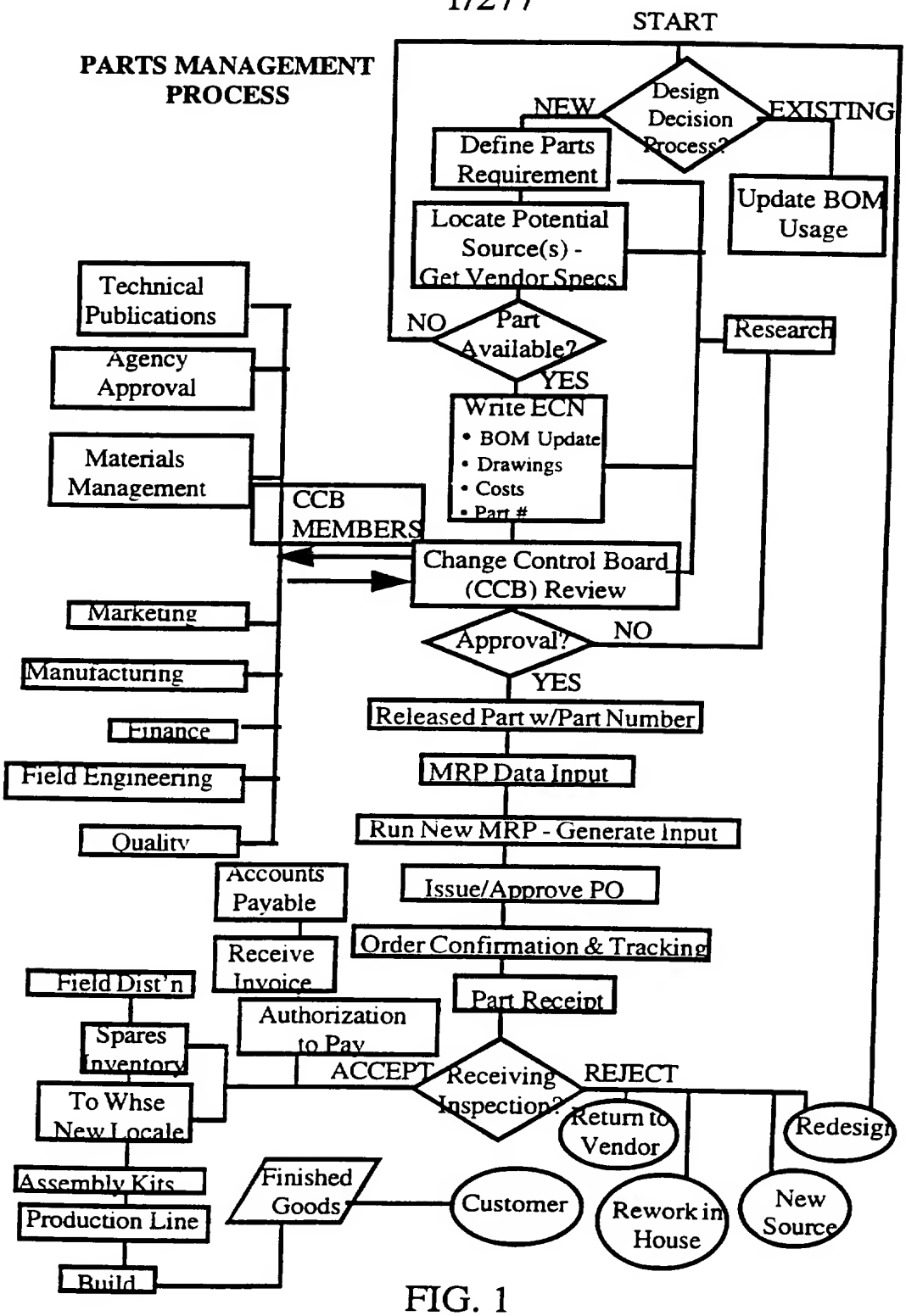

PARTS MANAGEMENT PROCESS
START
Design Decision Process?
NEW
EXISTING
Define Parts Requirement
Update BOM Usage
Locate Potential Source(s) - Get Vendor Specs
Technical Publications
Agency Approval
Materials Management
NO
Part Available?
YES
Research
Write ECN
• BOM Update
• Drawings
• Costs
• Part #
CCB MEMBERS
Change Control Board (CCB) Review
Marketing
Manufacturing
Finance
Field Engineering
Quality
Approval?
NO
YES
Released Part w/Part Number
MRP Data Input
Run New MRP - Generate Input
Accounts Payable
Issue/Approve PO
Order Confirmation & Tracking
Receive Invoice
Part Receipt
Field Dist'n
Authorization to Pay
Spares Inventory
ACCEPT
Receiving Inspection?
REJECT
To Whse New Locale
Return to Vendor
Redesign
Assembly Kits
Finished Goods
Customer
Rework in House
New Source
Production Line
Build

FIG. 1

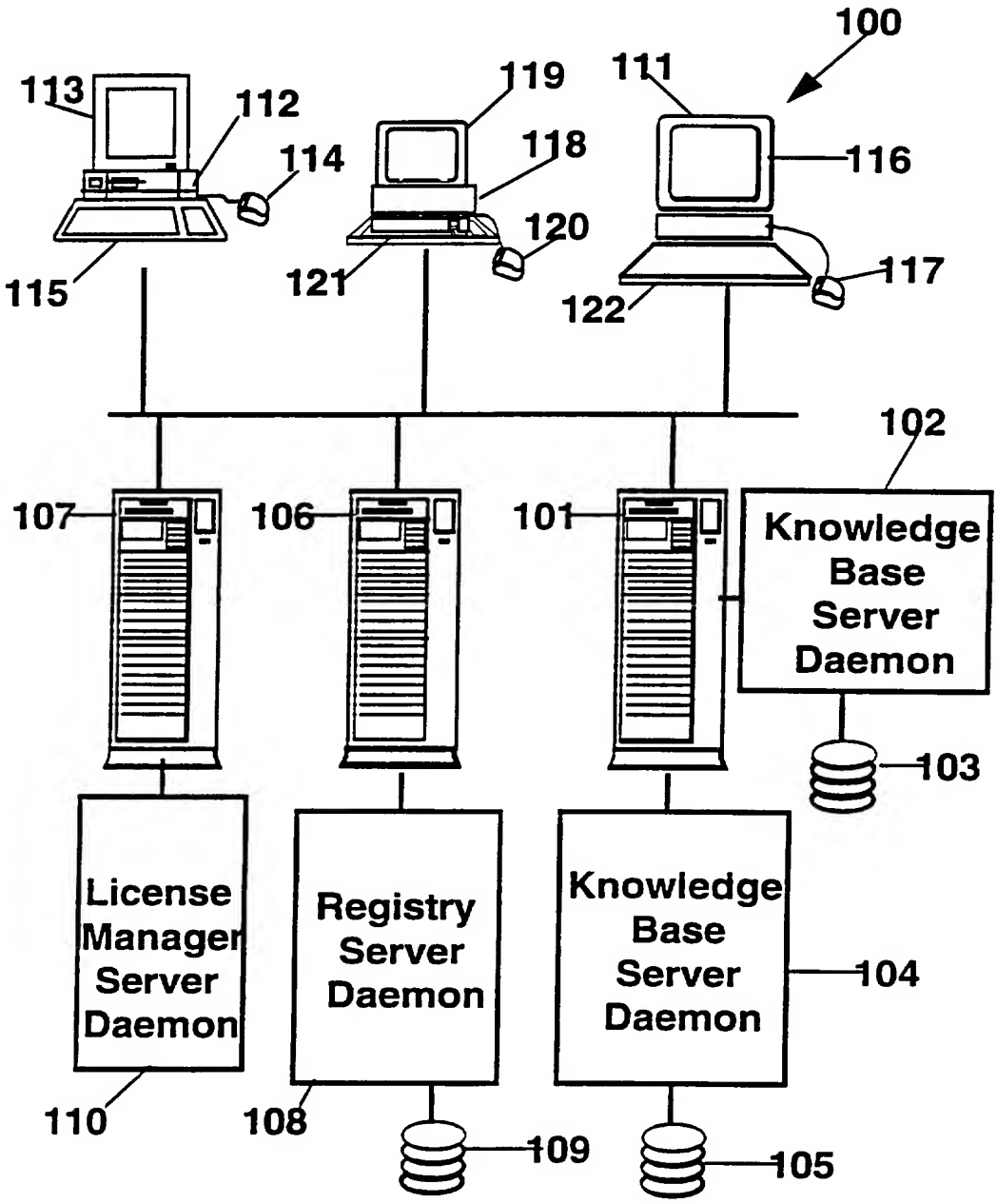

100
111
113
112
119
118
114
116
120
115
121
122
117
102
107
106
101
Knowledge Base Server Daemon
103
License Manager Server Daemon
Registry Server Daemon
Knowledge Base Server Daemon
104
110
108
109
105

FIG. 2

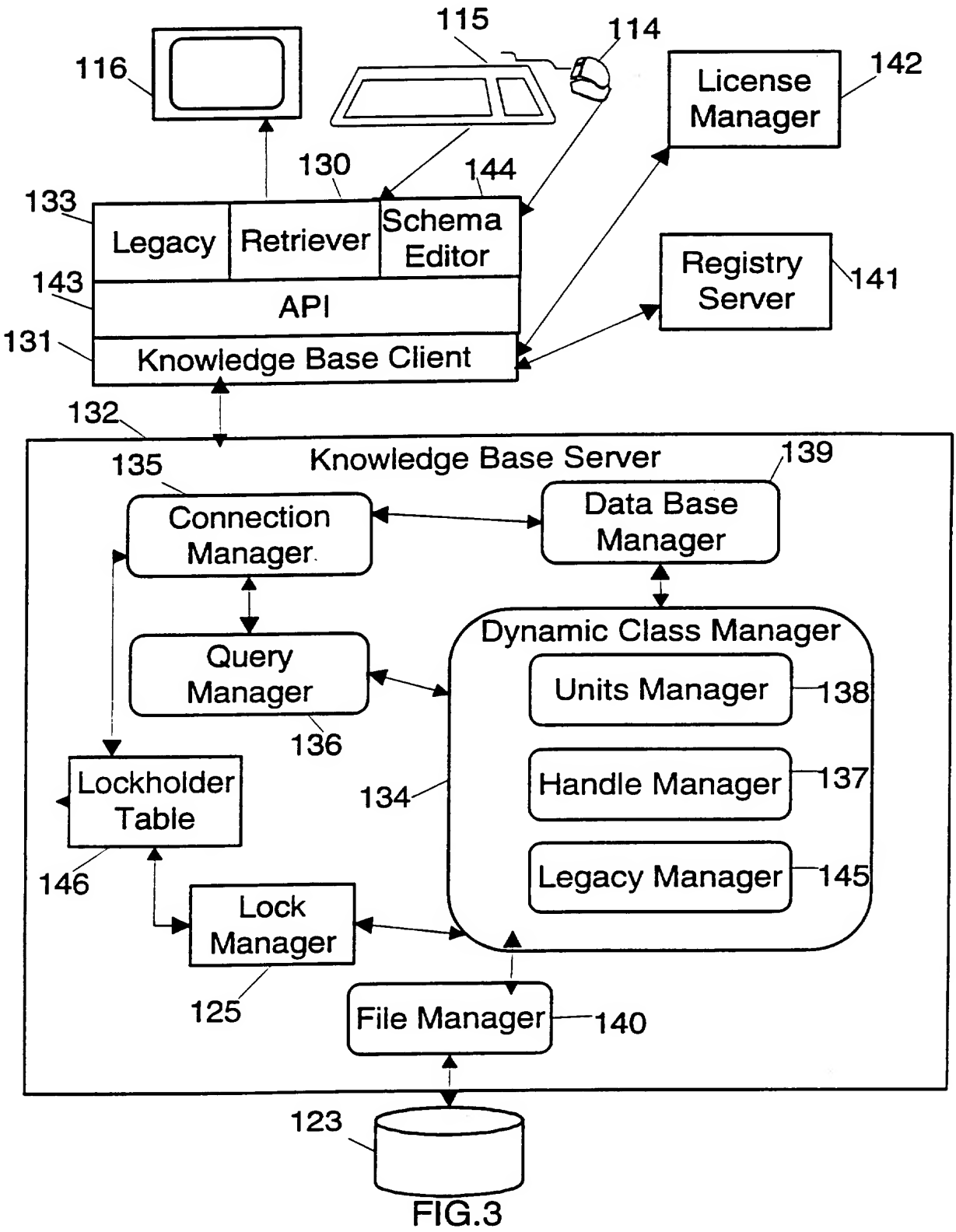

115
114
116
License Manager
142
130
144
133
Legacy
Retriever
Schema Editor
143
API
Registry Server
141
131
Knowledge Base Client
132
135
Knowledge Base Server
139
Connection Manager
Data Base Manager
Dynamic Class Manager
Query Manager
Units Manager
138
136
Lockholder Table
134
Handle Manager
137
Legacy Manager
145
146
Lock Manager
125
File Manager
140
123

FIG.3

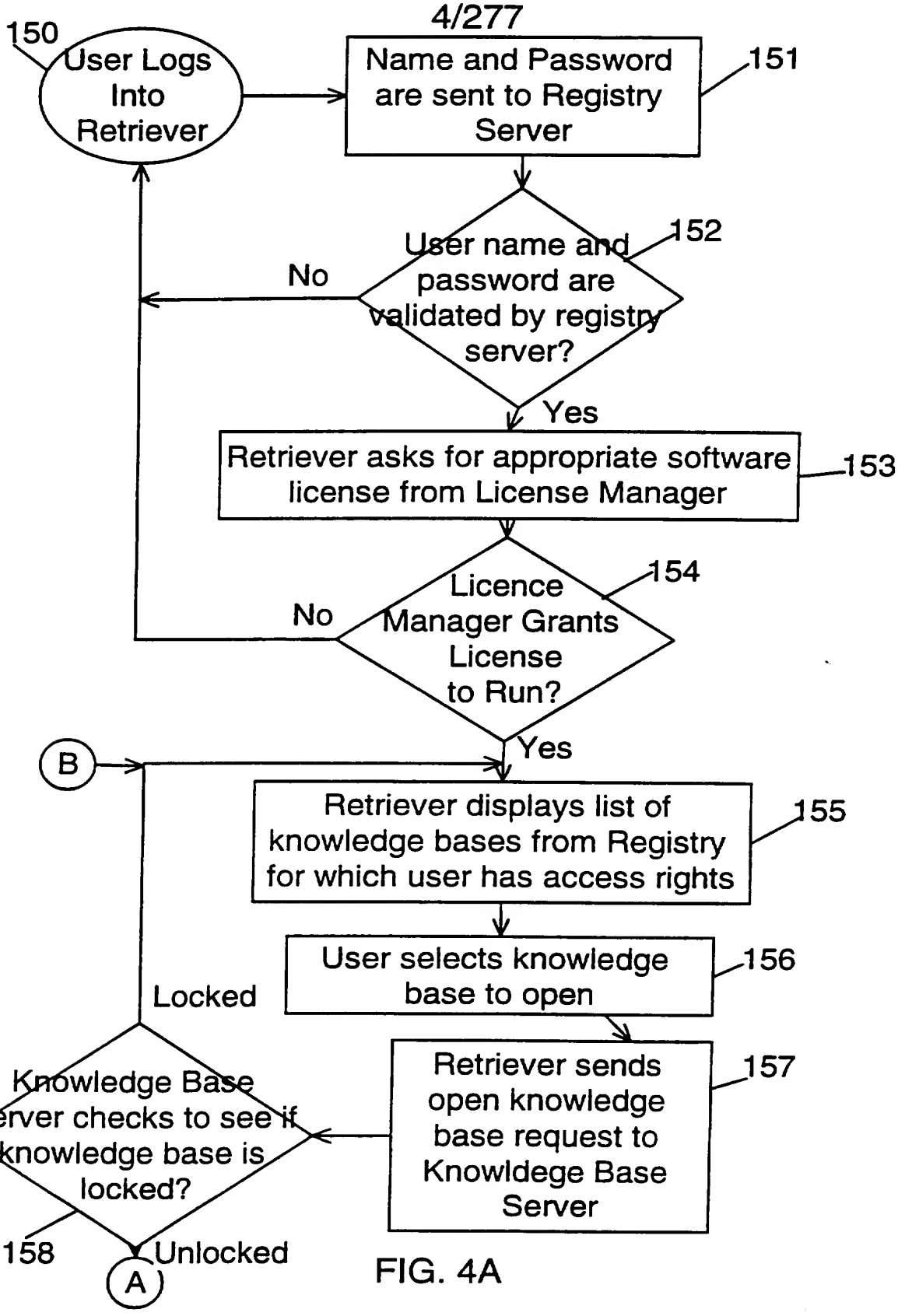

150
User Logs Into Retriever
Name and Password are sent to Registry Server
151
User name and password are validated by registry server?
152
No
Yes
Retriever asks for appropriate software license from License Manager
153
Licence Manager Grants License to Run?
154
No
Yes
B
Retriever displays list of knowledge bases from Registry for which user has access rights
155
User selects knowledge base to open
156
Locked
Retriever sends open knowledge base request to Knowldege Base Server
157
Knowledge Base Server checks to see if knowledge base is locked?
158
Unlocked
A

FIG. 4A

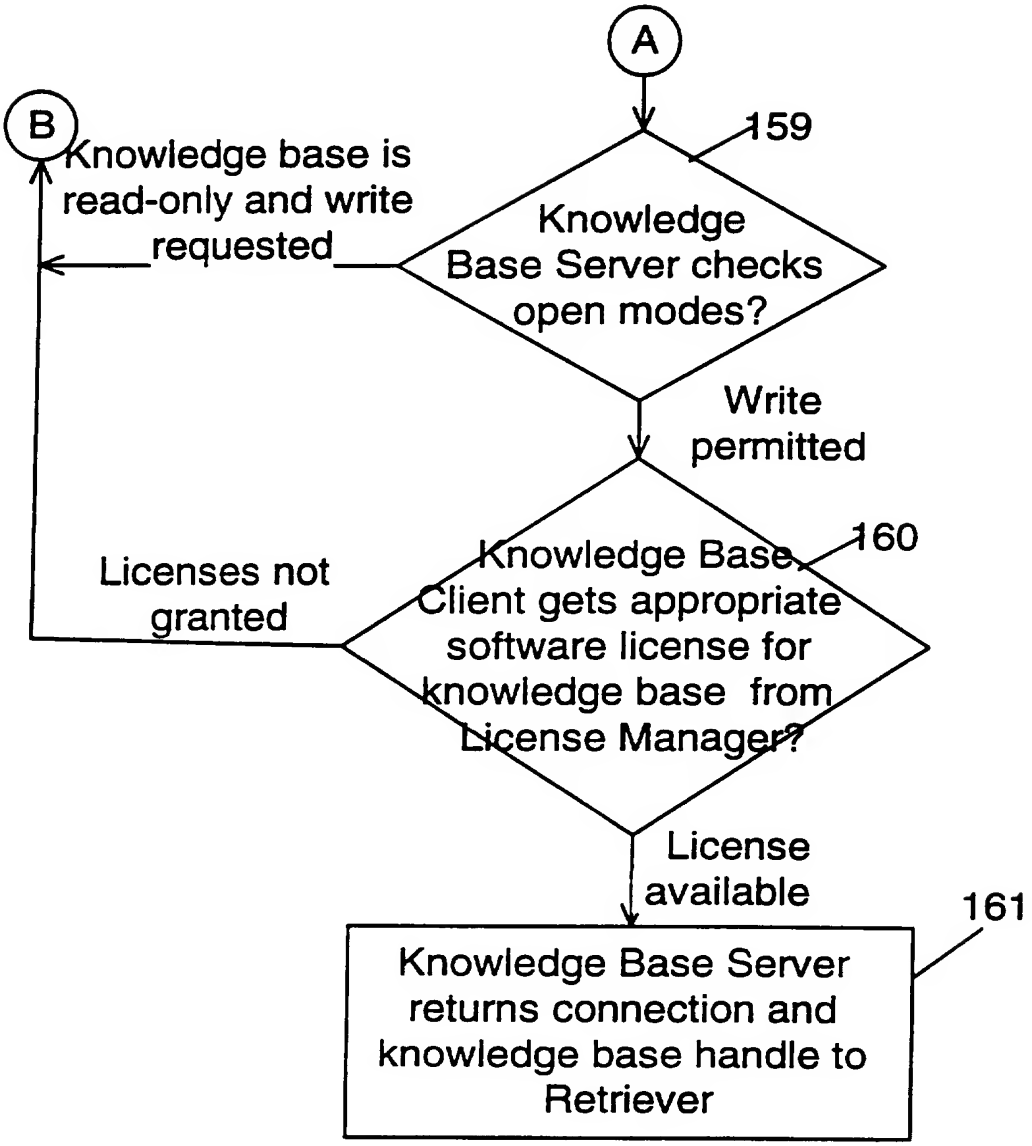

A
B
159
Knowledge base is read-only and write requested
Knowledge Base Server checks open modes?
Write permitted
160
Licenses not granted
Knowledge Base Client gets appropriate software license for knowledge base from License Manager?
License available
161
Knowledge Base Server returns connection and knowledge base handle to Retriever

FIG. 4B

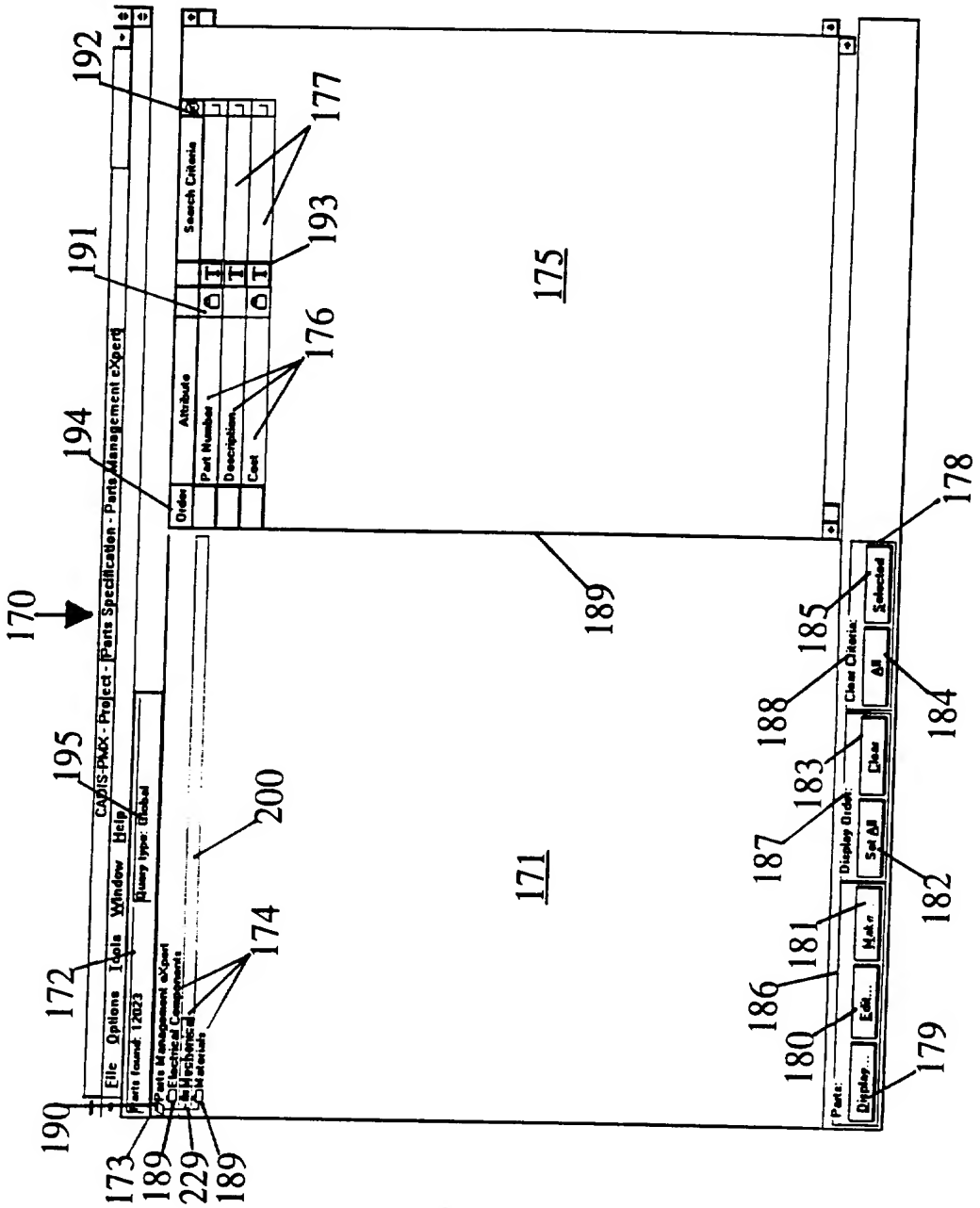

192
177
193
191
175
176
194
178
170
189
185
188
184
195
200
183
187
182
171
174
172
181
186
180
179
190
173
189
229
189
CADIS-PMX - Project - [Parts Specification - Parts Management eXpert]
File Options Tools Window Help
Search Criteria
Attribute
Part Number
Description
Cost
Order
Clear Criteria:
All
Selected
Clear
Display Order:
Set All
Maker
Edit...
Display...
Parts:

FIG. 5

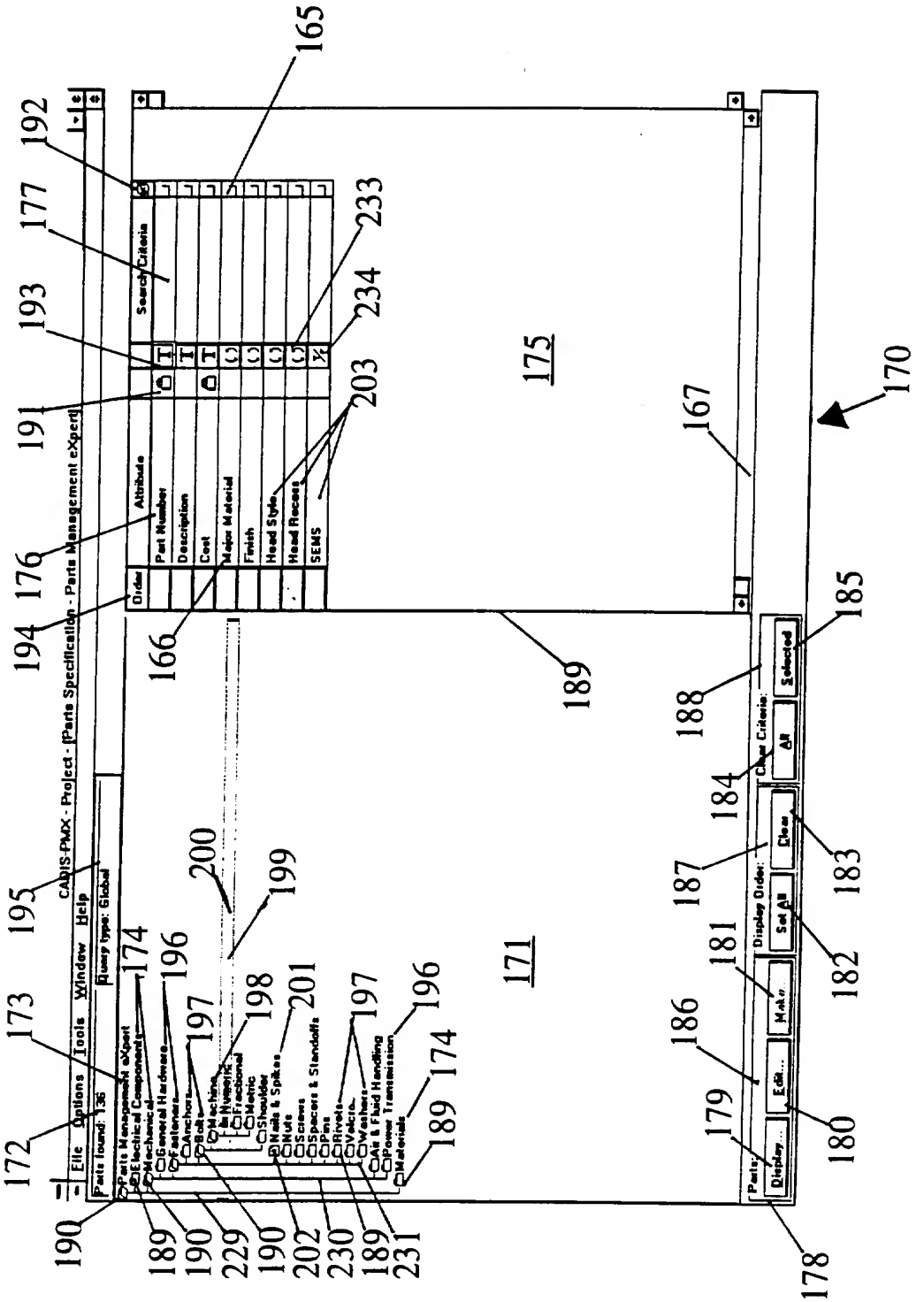

CADIS-PMX - Project - [Parts Specification - Parts Management eXpert]
File Options Tools Window Help
Parts found: 136
Query type: Global
Parts Management eXpert
Electrical Components
Mechanical
General Hardware
Fasteners
Anchors
Bolts
Machine
Rivnuts
Fractional
Metric
Shoulder
Nails & Spikes
Nuts
Screws
Spacers & Standoffs
Pins
Rivets
Valves
Washers
Air & Fluid Handling
Power Transmission
Materials
Order
Attribute
Part Number
Description
Cost
Major Material
Finish
Head Style
Head Recess
SEMS
Search Criteria
Clear Criteria:
All
Selected
Display Order:
Clear
Set All
Make
Edit...
Parts:
Display...
165
166
167
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
229
230
231
233
234

FIG. 6

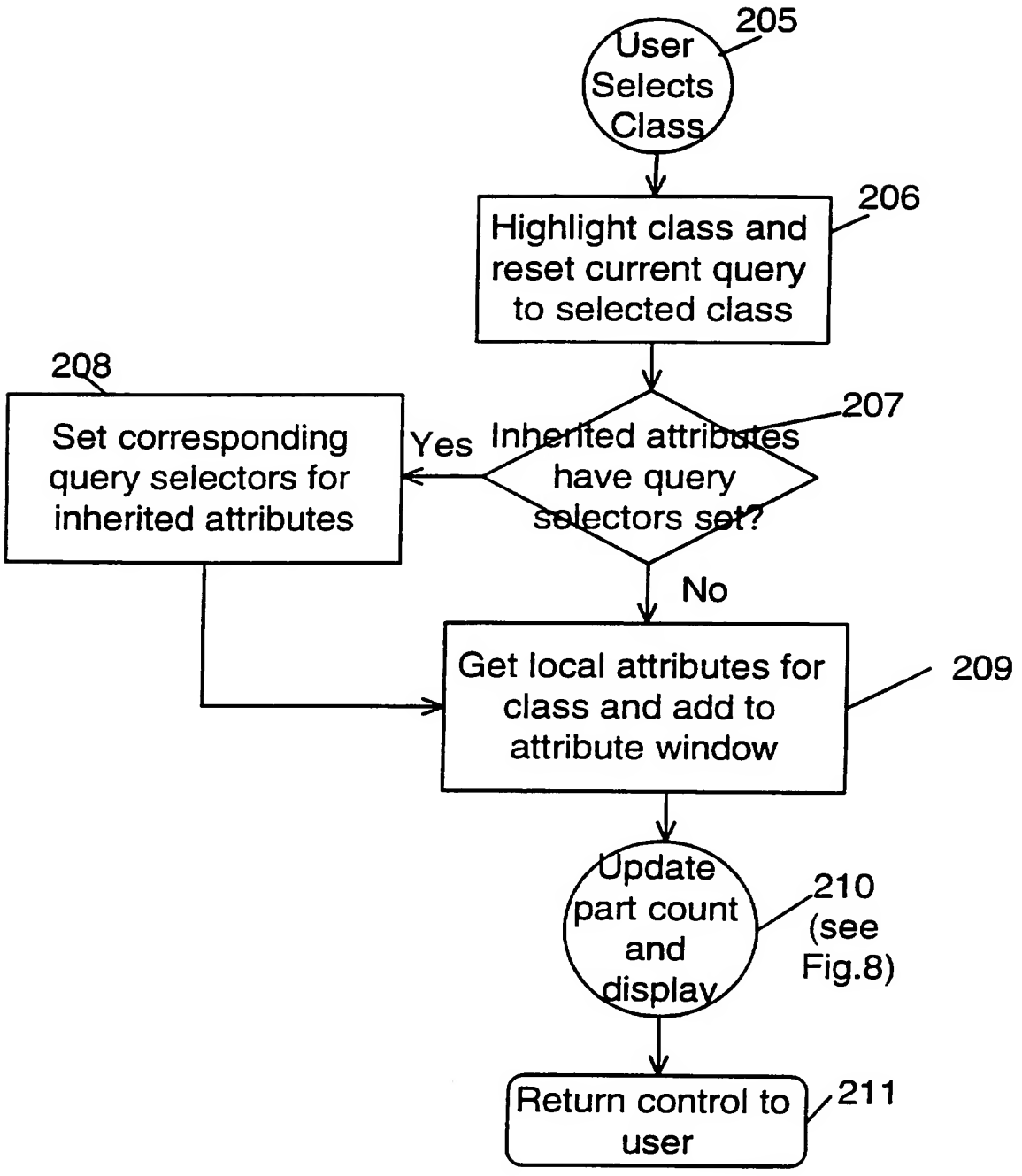

205
User Selects Class
206
Highlight class and reset current query to selected class
207
Inherited attributes have query selectors set?
Yes
208
Set corresponding query selectors for inherited attributes
No
209
Get local attributes for class and add to attribute window
Update part count and display
210 (see Fig.8)
Return control to user
211

FIG. 7

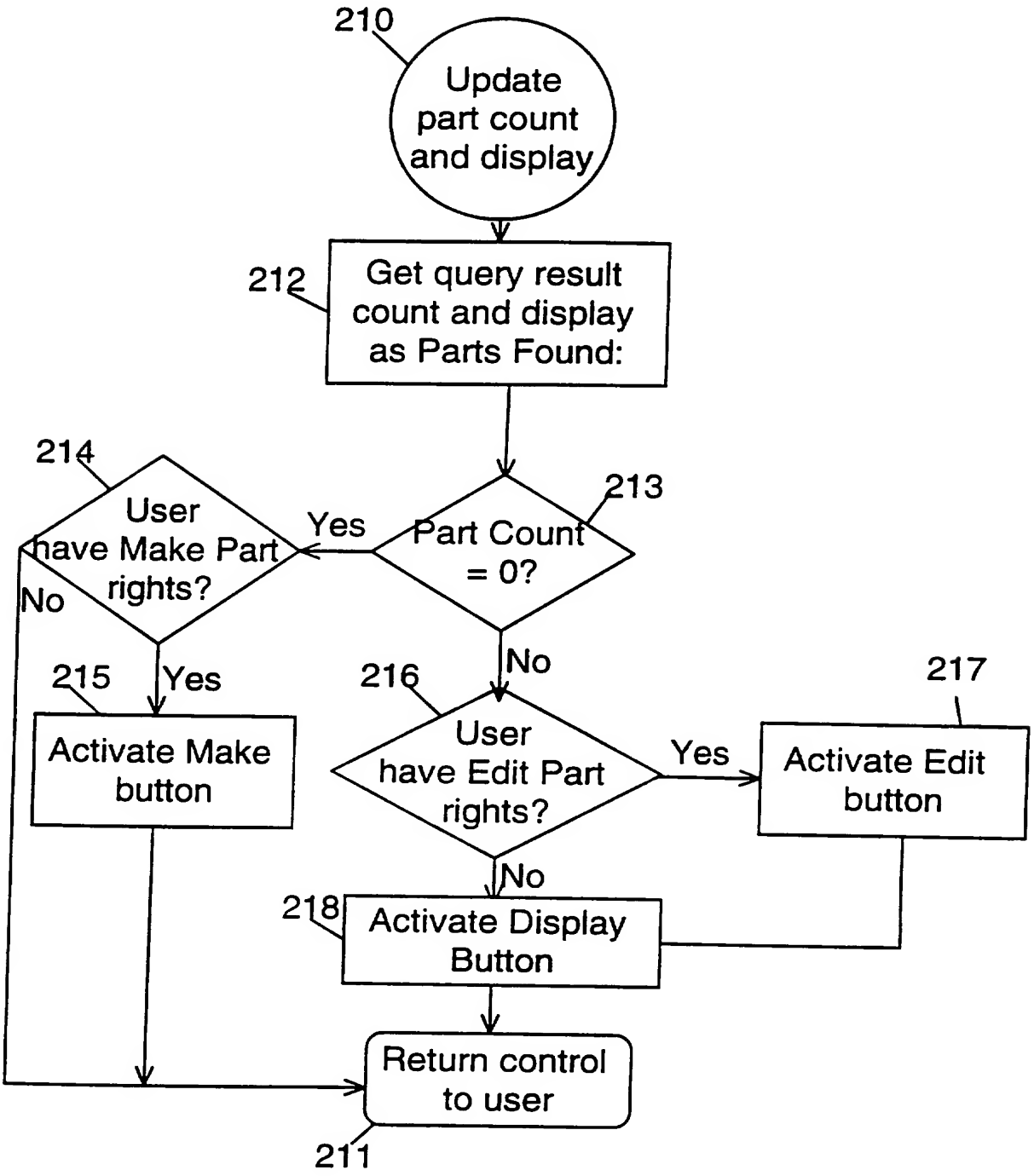

210
Update part count and display
212
Get query result count and display as Parts Found:
213
Part Count = 0?
Yes
214
User have Make Part rights?
No
215
Yes
Activate Make button
No
216
User have Edit Part rights?
Yes
217
Activate Edit button
No
218
Activate Display Button
Return control to user
211

FIG. 8

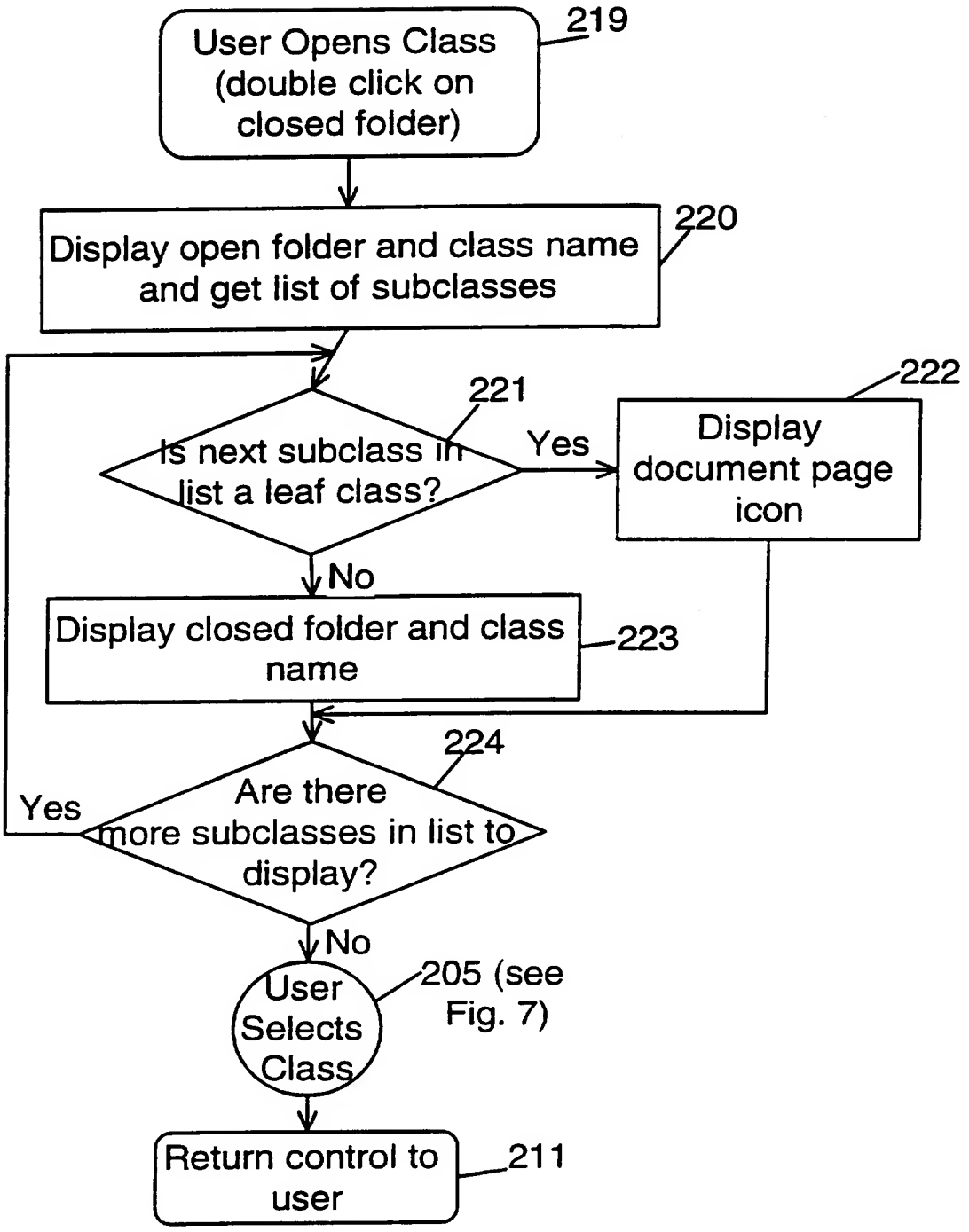

User Opens Class (double click on closed folder)
219
Display open folder and class name and get list of subclasses
220
221
Is next subclass in list a leaf class?
Yes
222
Display document page icon
No
Display closed folder and class name
223
224
Are there more subclasses in list to display?
Yes
No
User Selects Class
205 (see Fig. 7)
Return control to user
211

FIG. 9

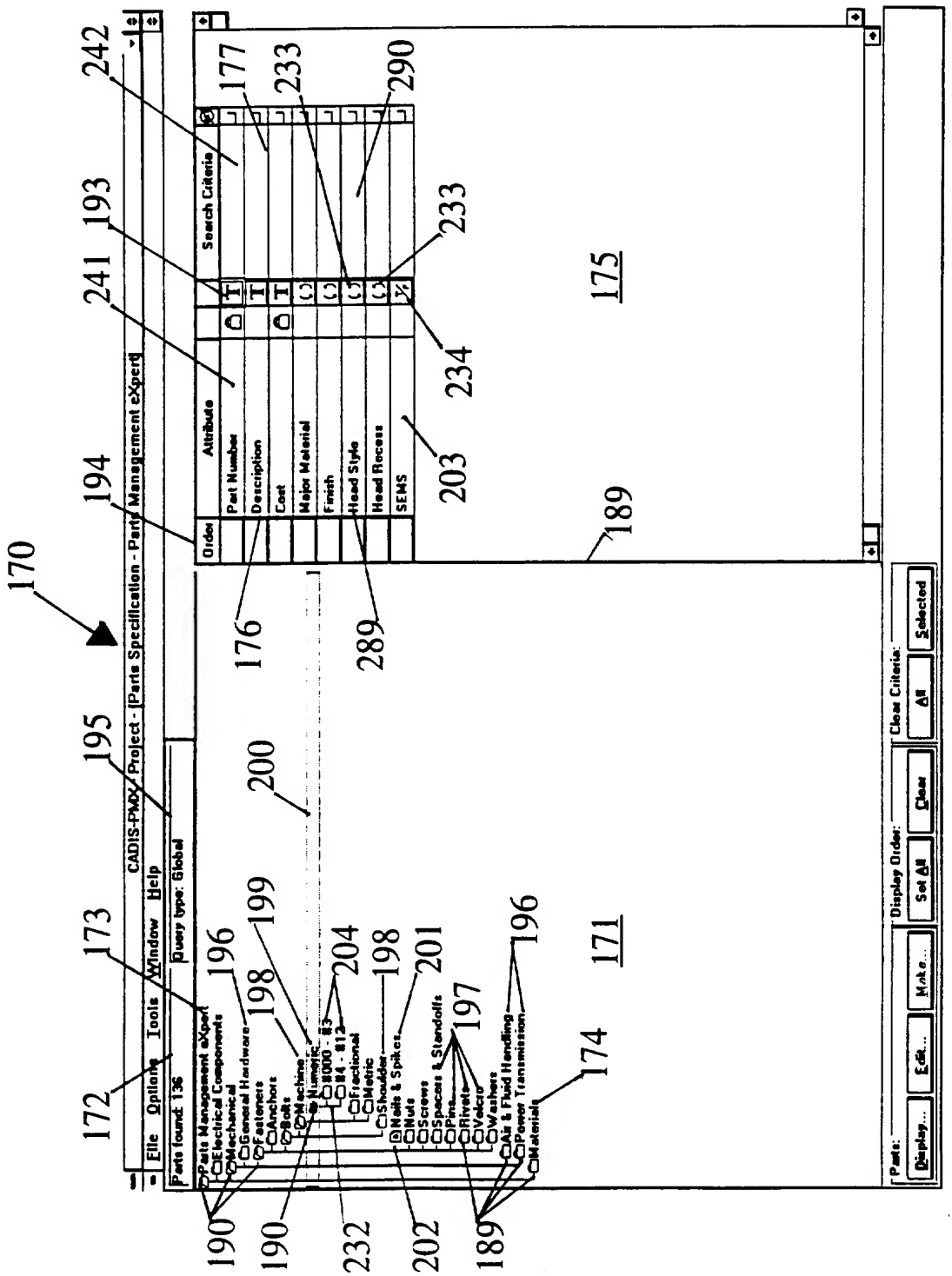

170
172
173
174
175
176
177
189
190
190
194
195
196
196
197
198
198
199
200
201
202
203
204
232
233
233
234
241
242
289
290
CADIS-PMX Project - [Parts Specification - Parts Management eXpert]
File Options Tools Window Help
Parts found: 136
Query type: Global
Parts Management eXpert
Electrical Components
Mechanical
General Hardware
Fasteners
Anchors
Bolts
Machine
Numeric
#000 - #3
#4 - #12
Fractional
Metric
Shoulder
Nuts & Spikes
Nuts
Screws
Spacers & Standoffs
Pins
Rivets
Velcro
Washers
Air & Fluid Handling
Power Transmission
Material
Search Criteria
Order
Attribute
Part Number
Description
Cost
Major Material
Finish
Head Style
Head Recess
SEMS
Parts:
Display...
Edit...
Make...
Display Order:
Set All
Clear Criteria:
Clear
All
Selected

FIG. 10

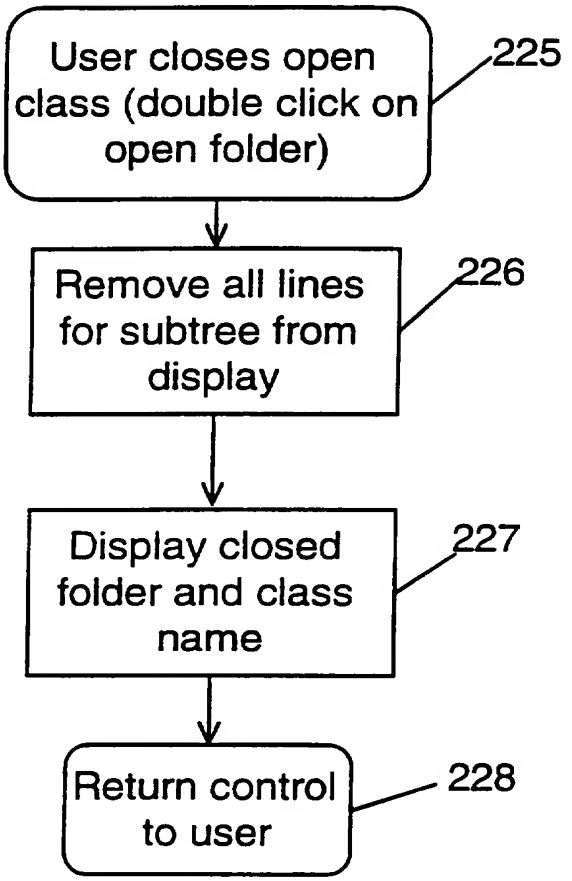

User closes open class (double click on open folder)
225
Remove all lines for subtree from display
226
Display closed folder and class name
227
Return control to user
228

FIG. 11

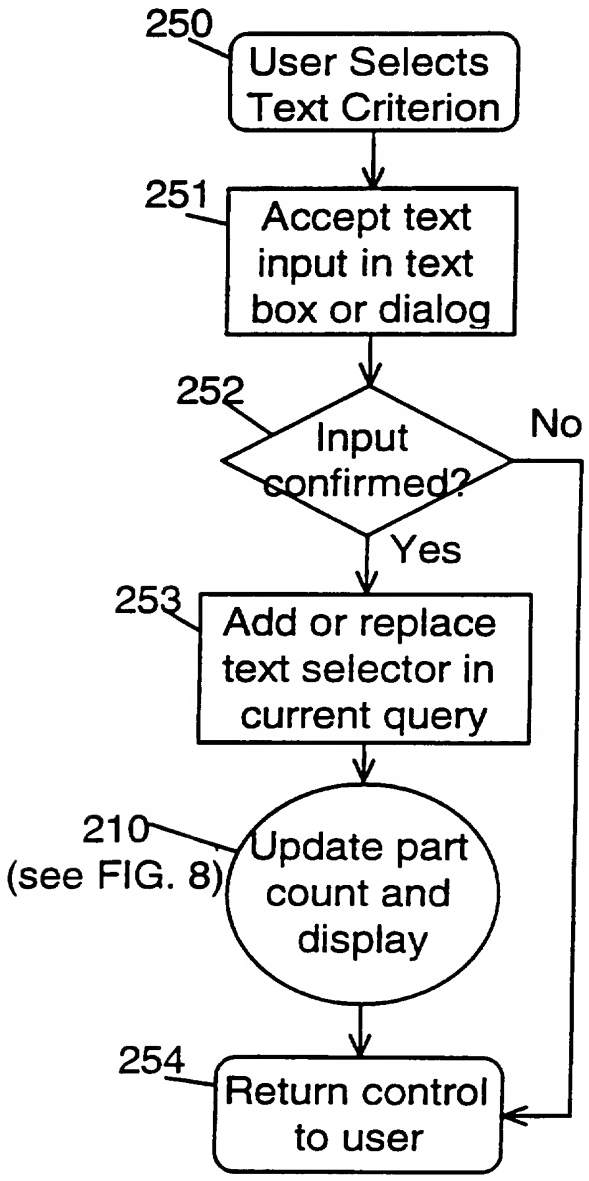

250
User Selects Text Criterion
251
Accept text input in text box or dialog
252
Input confirmed?
No
Yes
253
Add or replace text selector in current query
210
(see FIG. 8)
Update part count and display
254
Return control to user

FIG. 12

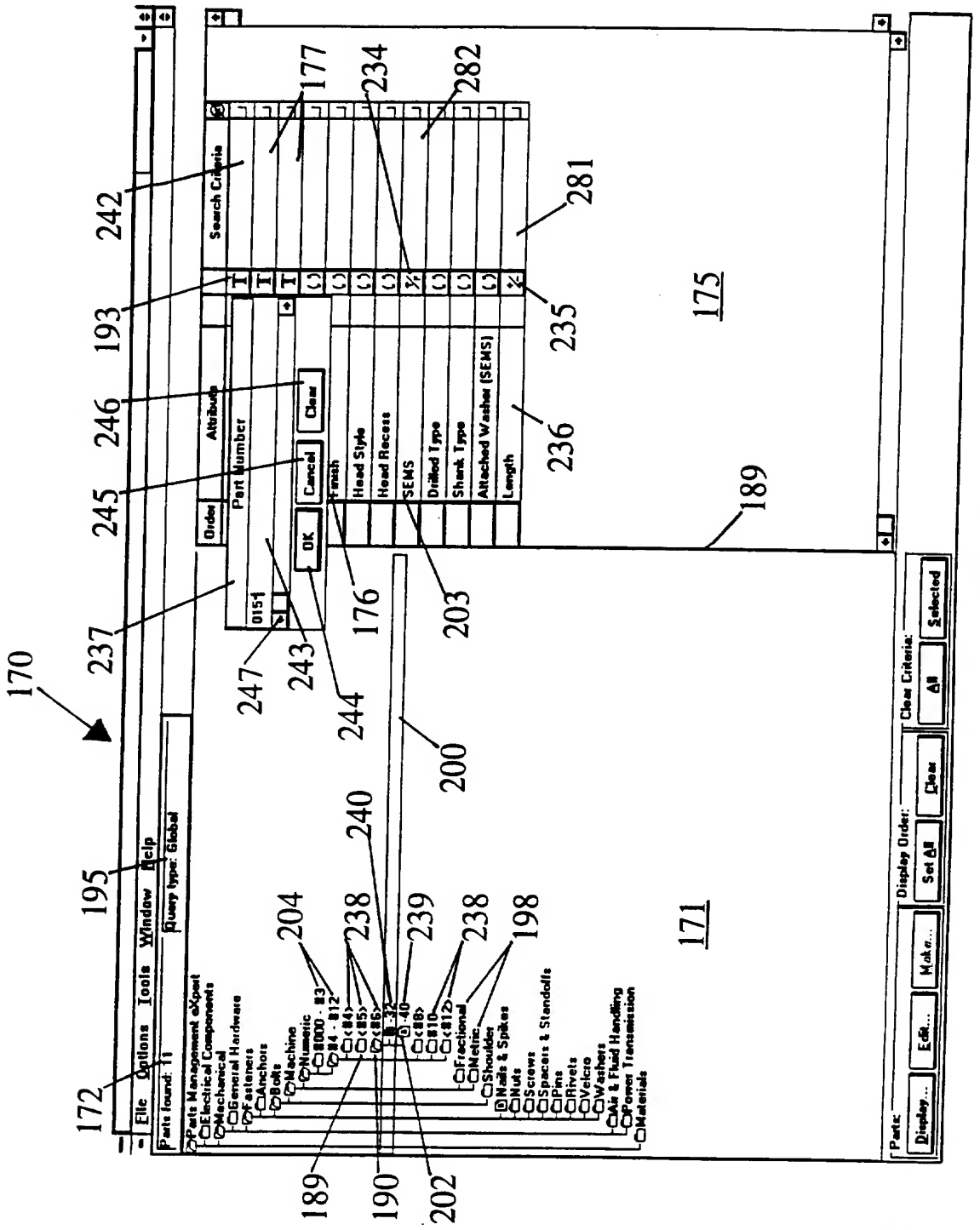

170
172
195
237
247
243
244
200
240
204
238
239
238
198
189
190
202
171
175
189
176
203
236
246
245
242
193
177
234
282
281
235
Search Criteria
Attribute
Part Number
Order
OK
Cancel
Clear
Finish
Head Style
Head Recess
SEMS
Drilled Type
Shank Type
Attached Washer (SEMS)
Length
File
Options
Tools
Window
Help
Parts found: 11
Query type: Global
Parts Management eXpert
Electrical Components
Mechanical
General Hardware
Fasteners
Anchors
Bolts
Machine
Numeric
Fractional
Metric
Shoulder
Nails & Spikes
Nuts
Screws
Spacers & Standoffs
Pins
Rivets
Velcro
Washers
Air & Fluid Handling
Power Transmission
Materials
Parts
Display...
Edit...
Make...
Display Order:
Set All
Clear
Clear Criteria:
All
Selected

FIG. 13

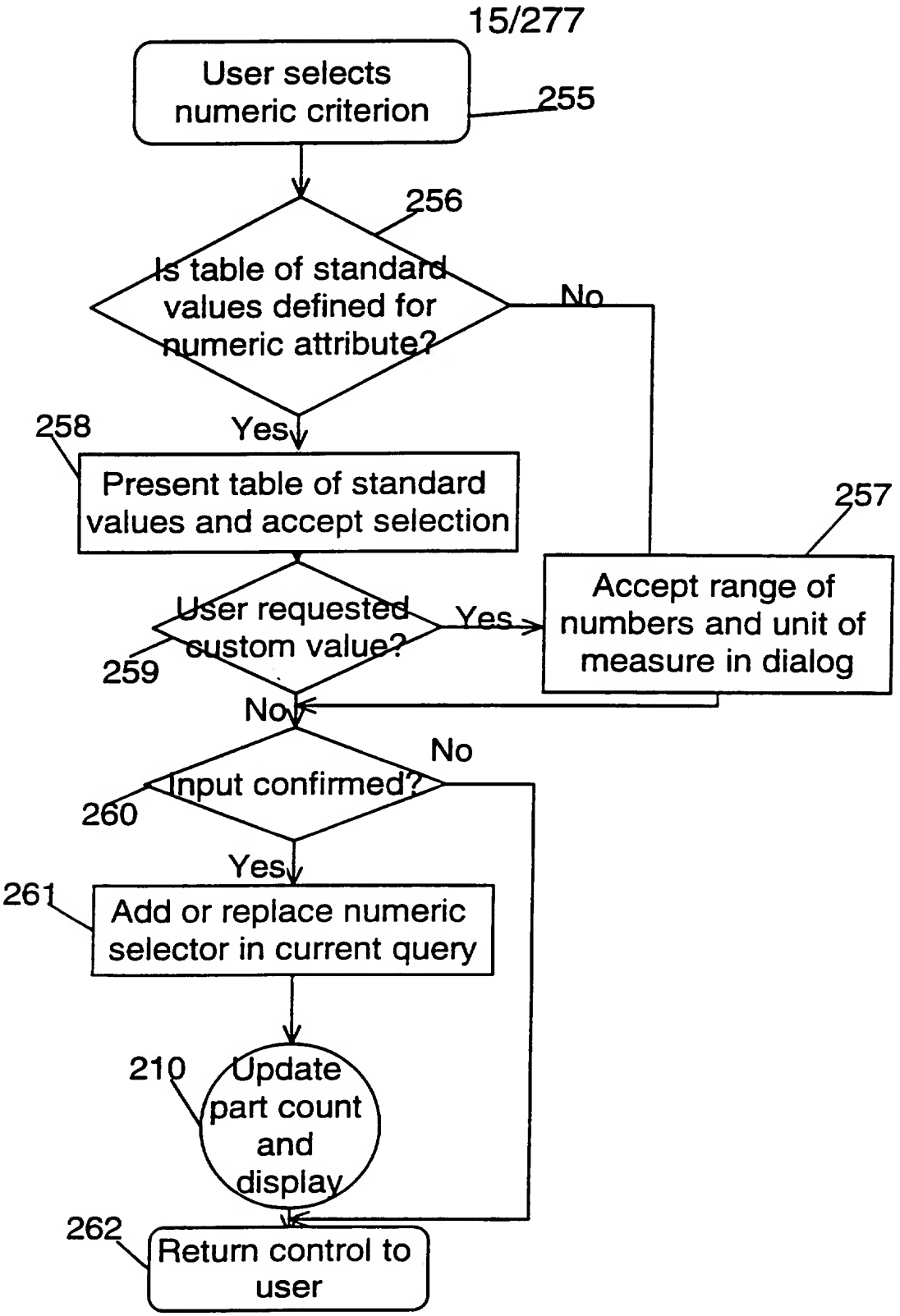

User selects numeric criterion
255
256
Is table of standard values defined for numeric attribute?
No
Yes
258
Present table of standard values and accept selection
257
Accept range of numbers and unit of measure in dialog
User requested custom value?
Yes
259
No
No
Input confirmed?
260
Yes
261
Add or replace numeric selector in current query
210
Update part count and display
262
Return control to user

FIG. 14

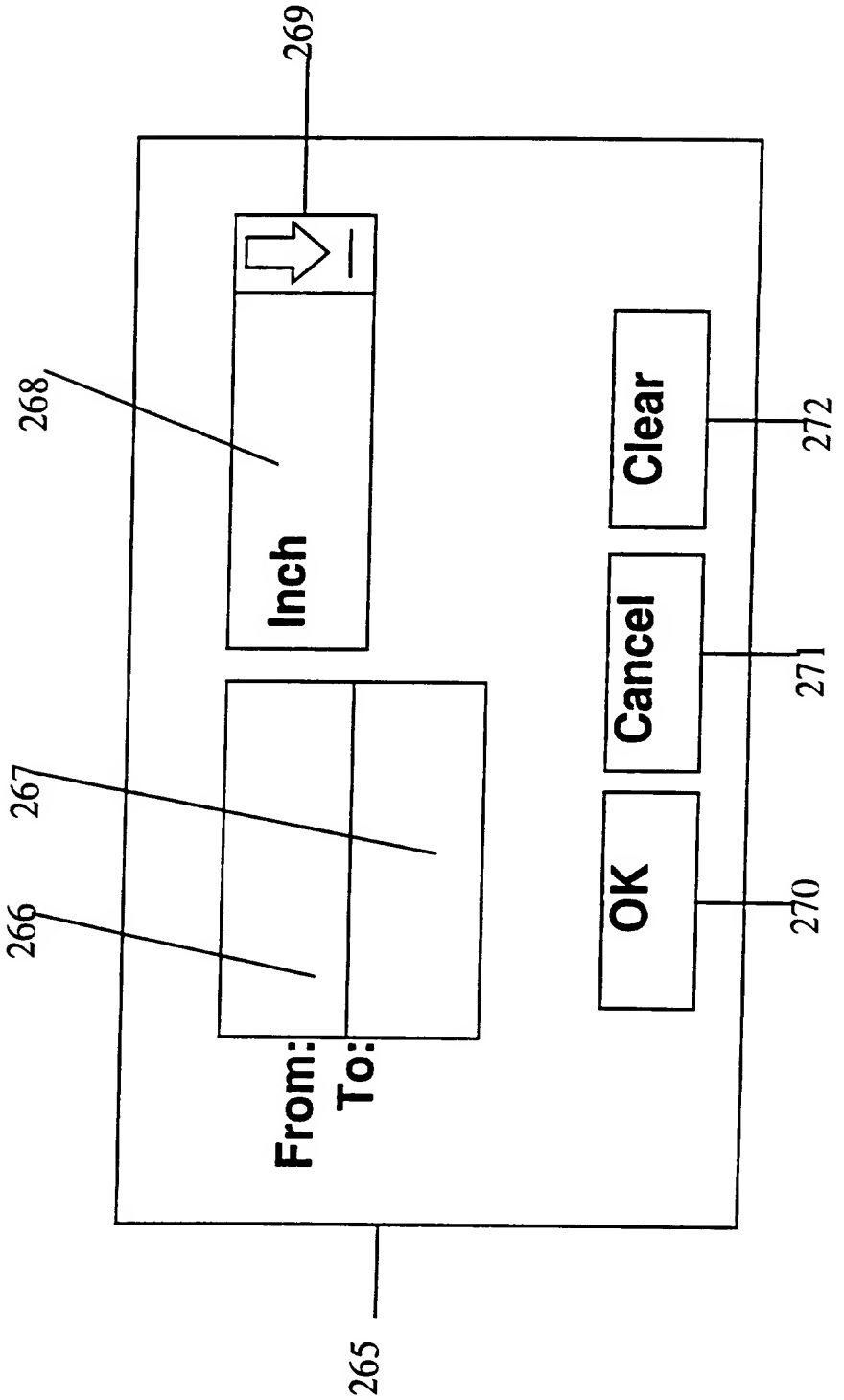


FIG. 15

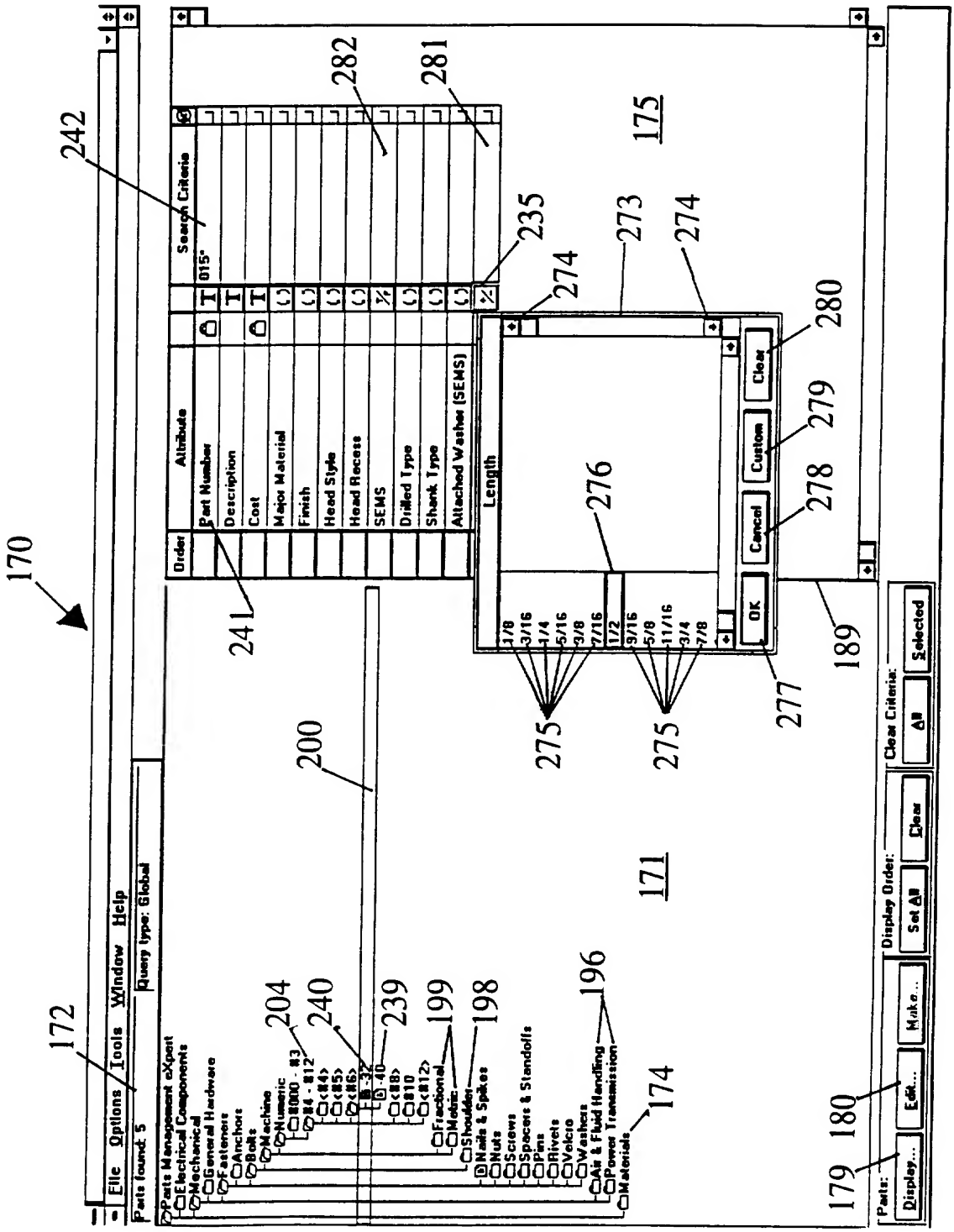

170
172
File Options Tools Window Help
Parts found: 5
Query type: Global
Parts Management eXpert
Electrical Components
Mechanical
General Hardware
Fasteners
Anchors
Bolts
Machine
Numeric
Fractional
Metric
Shoulder
Nails & Spikes
Nuts
Screws
Spacers & Standoffs
Pins
Rivets
Velcro
Washers
Air & Fluid Handling
Power Transmission
Materials
200
204
240
239
199
198
196
174
171
Order
Attribute
Search Criteria
Part Number
Description
Cost
Major Material
Finish
Head Style
Head Recess
SEMS
Drilled Type
Shank Type
Attached Washer (SEMS)
015*
241
242
282
281
235
273
274
175
Length
276
275
277
278
279
280
Clear
Custom
Cancel
OK
189
Clear Criteria:
All
Selected
Display Order:
Set All
Clear
Parts:
Display...
Edit...
Make...
179
180

FIG. 16

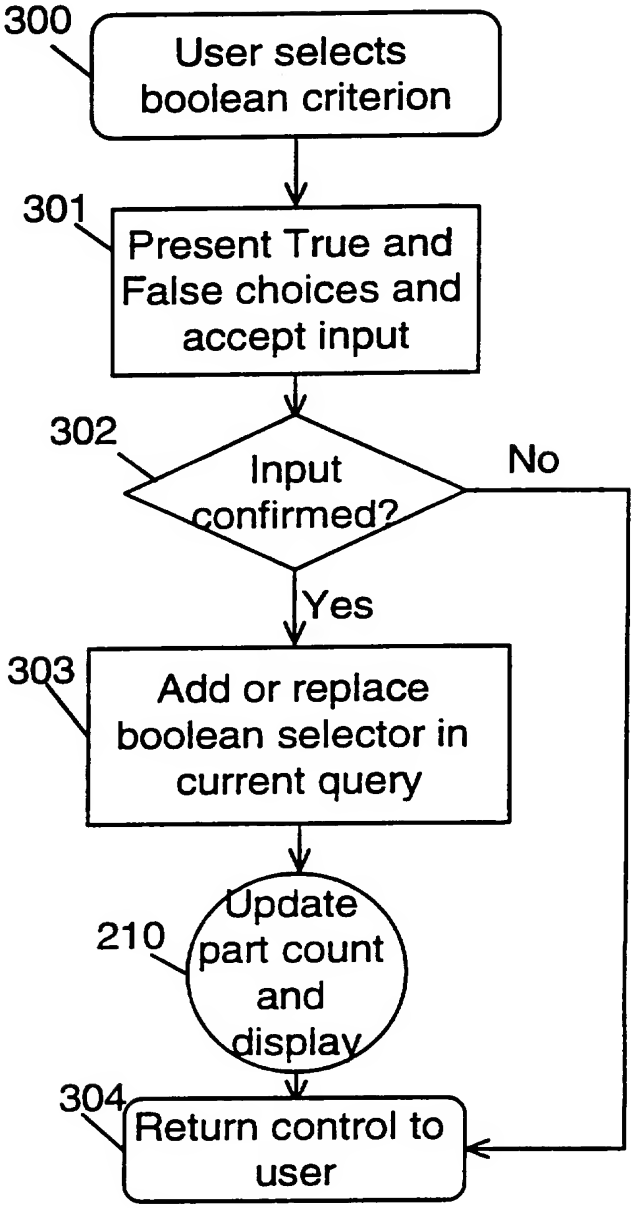

300
User selects boolean criterion
301
Present True and False choices and accept input
302
Input confirmed?
No
Yes
303
Add or replace boolean selector in current query
210
Update part count and display
304
Return control to user

FIG. 17

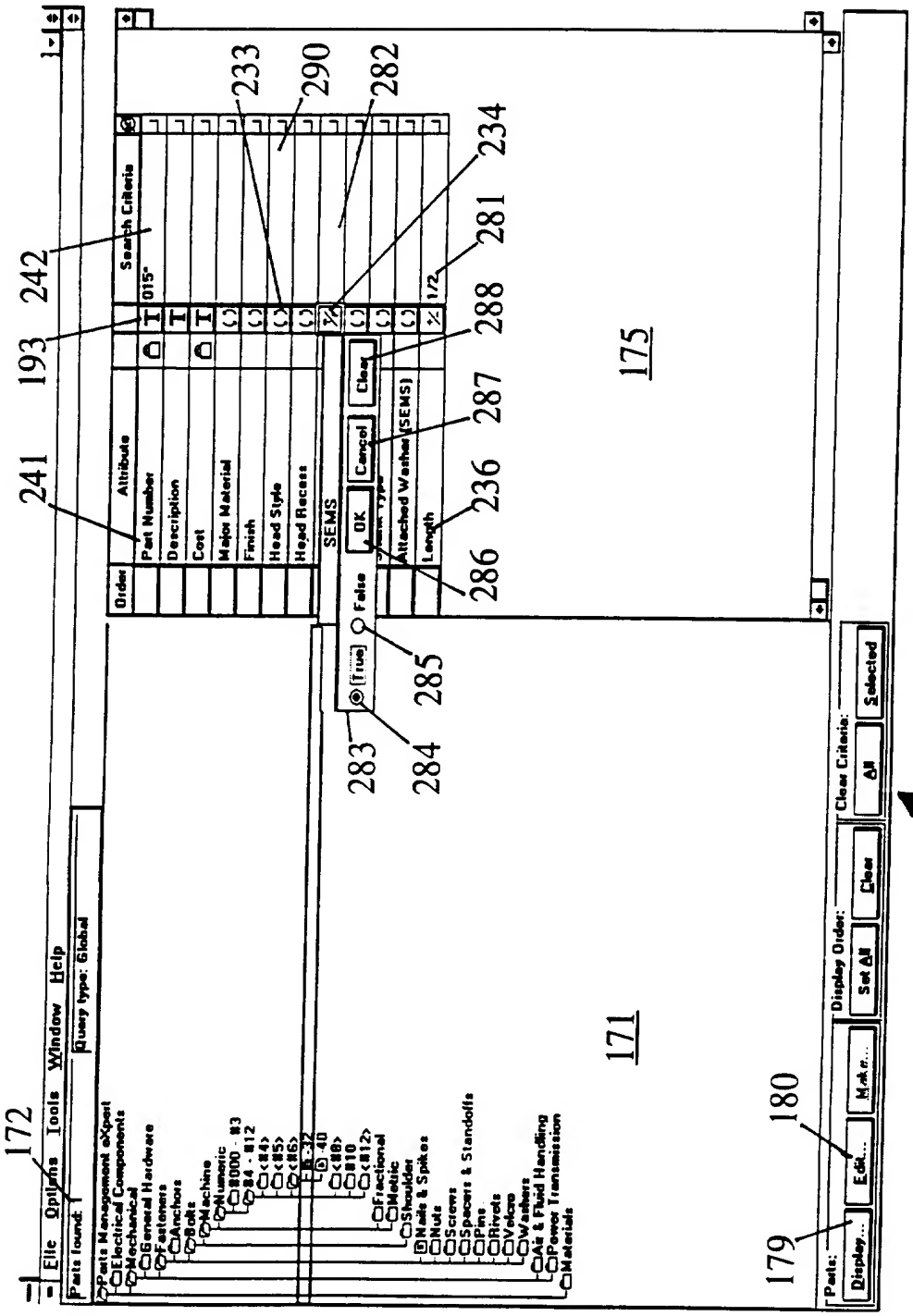

172
193
241
242
233
290
282
234
281
288
287
236
286
285
284
283
175
171
180
179
170
Search Criteria
Attribute
Order
Part Number
Description
Cost
Major Material
Finish
Head Style
Head Recess
Attached Washer [SEMS]
Length
SEMS
OK
Cancel
Clear
False
True
Parts:
Display...
Edit...
Make...
Display Order:
Set All
Clear
Clear Criteria:
All
Selected

FIG. 18

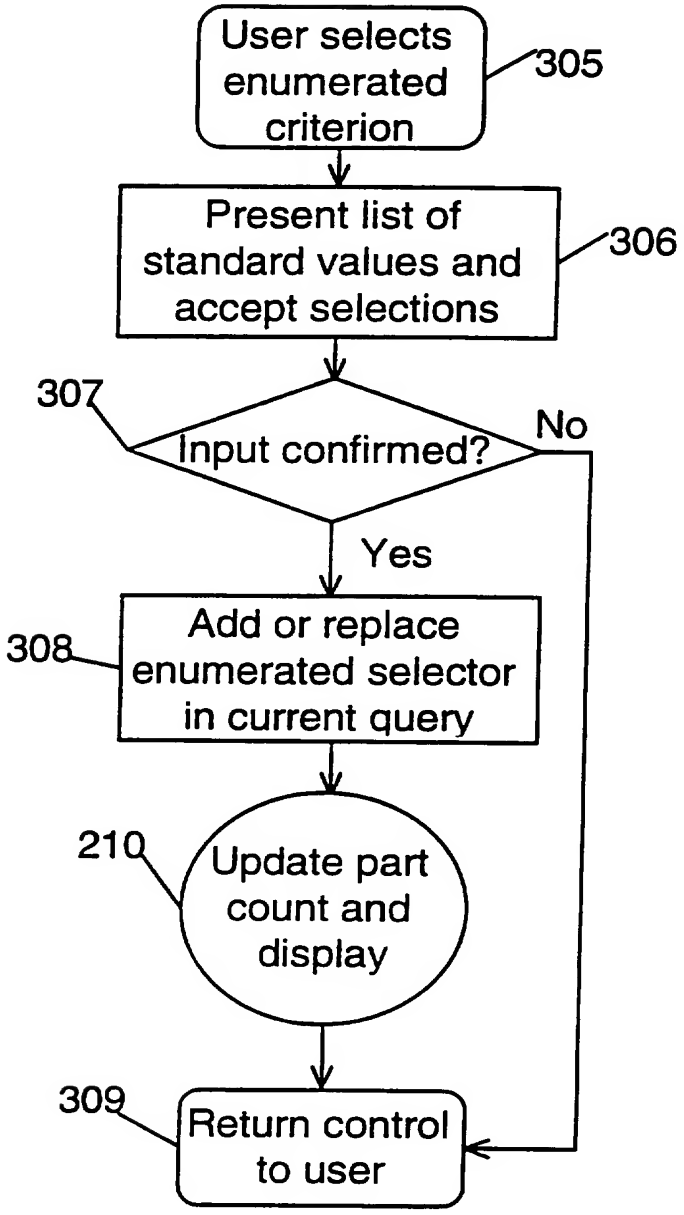

User selects enumerated criterion
305
Present list of standard values and accept selections
306
307
Input confirmed?
No
Yes
Add or replace enumerated selector in current query
308
210
Update part count and display
309
Return control to user

FIG. 19

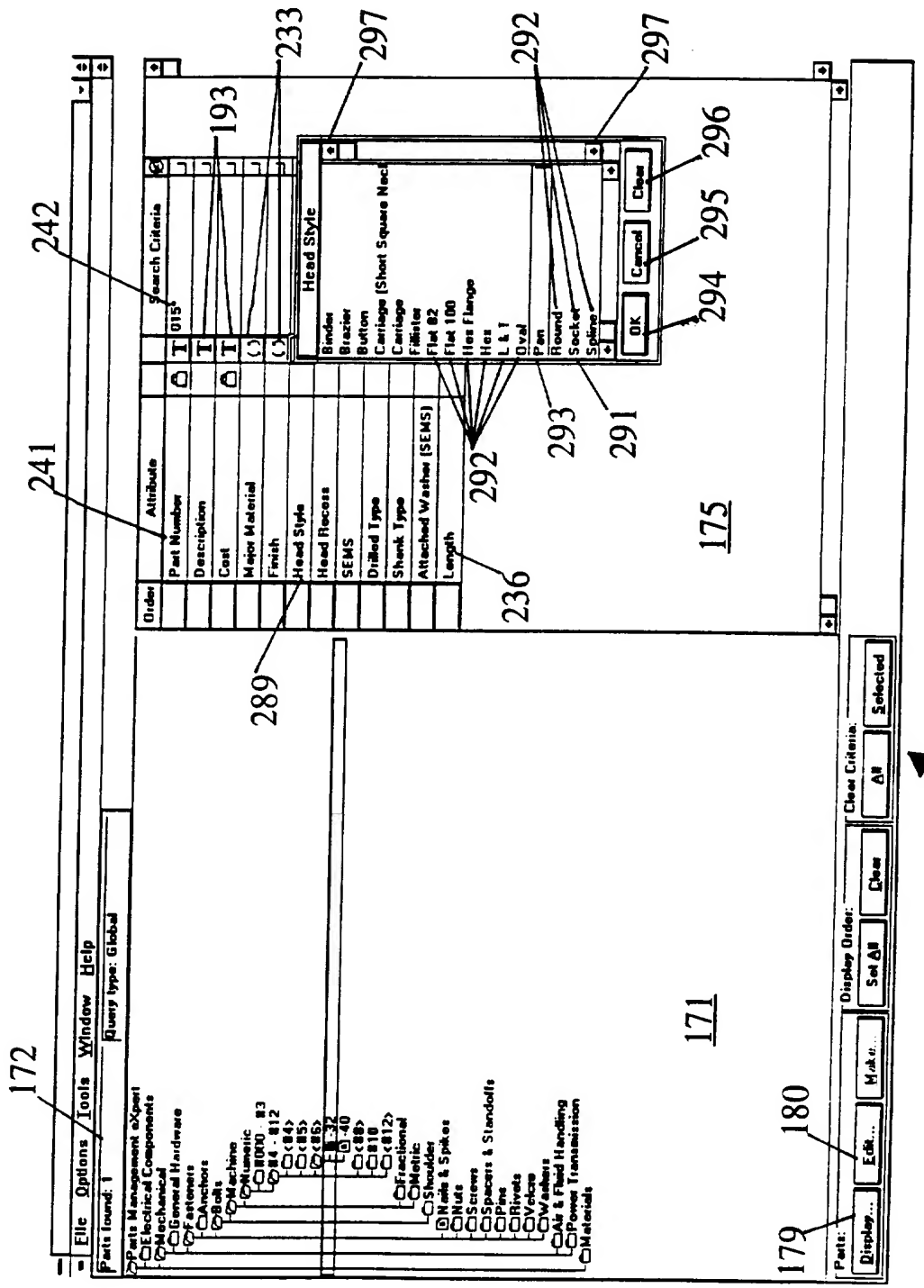

242
241
233
297
193
292
297
296
295
294
293
291
292
236
289
175
171
172
180
179
170
Search Criteria
Attribute
Order
Part Number
Description
Cost
Major Material
Finish
Head Style
Head Recess
SEMS
Drilled Type
Shank Type
Attached Washer (SEMS)
Length
Head Style
Binder
Brazier
Button
Carriage (Short Square Neck
Carriage
Fillister
Flat 82
Flat 100
Hex Flange
Hex
L & T
Oval
Pan
Round
Socket
Spline
OK
Cancel
Clear
Clear Criteria:
All
Selected
Display Order:
Sel All
Clear
Parts:
Display...
Edit...
Make...
Parts found: 1
Query type: Global
File Options Tools Window Help
Parts Management eXpert
Electrical Components
Mechanical
General Hardware
Fasteners
Anchors
Bolts
Machine
Numeric
Nuts & Spikes
Screws
Spacers & Standoffs
Pins
Rivets
Velcro
Washers
Air & Fluid Handling
Power Transmission
Materials

FIG. 20

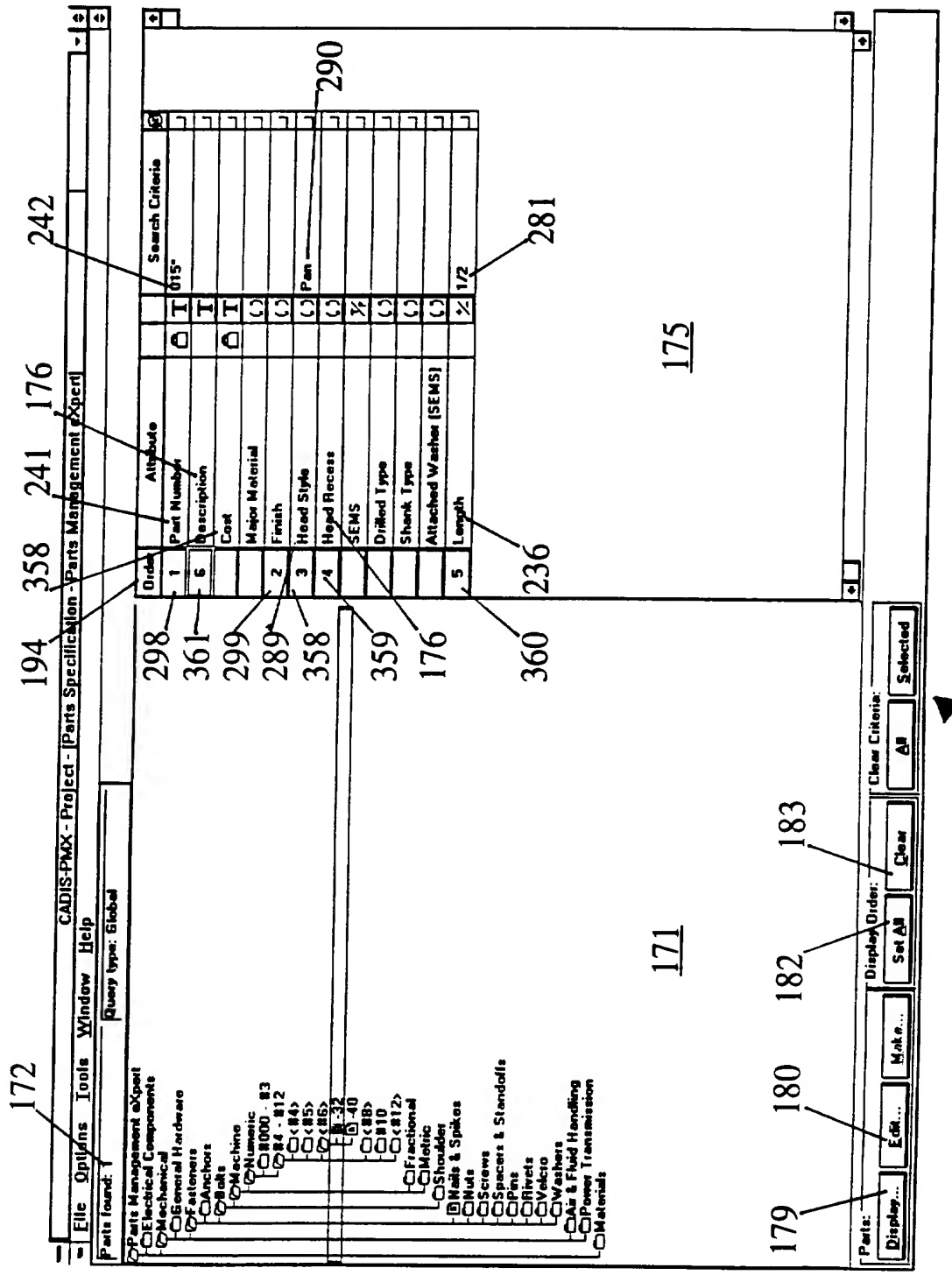

CADIS-PMX - Project - [Parts Specification - Parts Management eXpert]
File Options Tools Window Help
Parts found: 1
Query type: Global
Parts Management eXpert
Electrical Components
Mechanical
General Hardware
Fasteners
Anchors
Bolts
Machine
Numeric
#000 - #3
#4 - #12
<#4>
<#5>
<#6>
1/4 - 3/32
5/16 - 3/40
<#8>
#10
<#12>
Fractional
Metric
Shoulder
Nails & Spikes
Nuts
Screws
Spacers & Standoffs
Pins
Rivets
Velcro
Washers
Air & Fluid Handling
Power Transmission
Materials
Order
Attribute
Search Criteria
Part Number
Description
Cost
Major Material
Finish
Head Style
Head Recess
SEMS
Drilled Type
Shank Type
Attached Washer (SEMS)
Length
015"
Pan
1/2
Parts:
Display...
Edit...
Make...
Display Order:
Set All
Clear Criteria:
Clear
All
Selected
194
358
241
176
242
290
281
175
236
298
361
299
289
358
359
176
360
171
172
180
182
183
179
170

FIG. 21

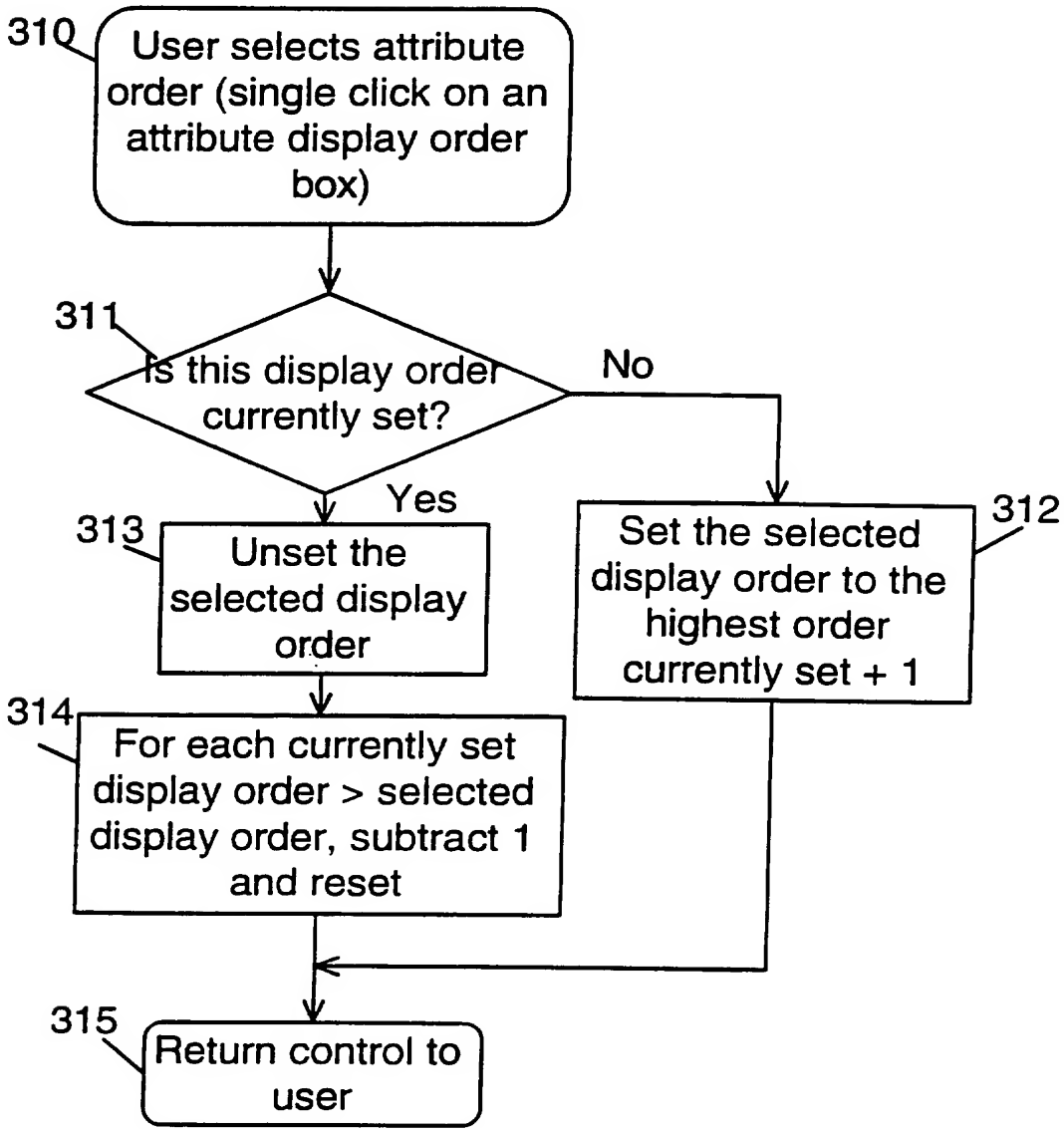

310
User selects attribute order (single click on an attribute display order box)
311
Is this display order currently set?
No
Yes
312
Set the selected display order to the highest order currently set + 1
313
Unset the selected display order
314
For each currently set display order > selected display order, subtract 1 and reset
315
Return control to user

FIG. 22

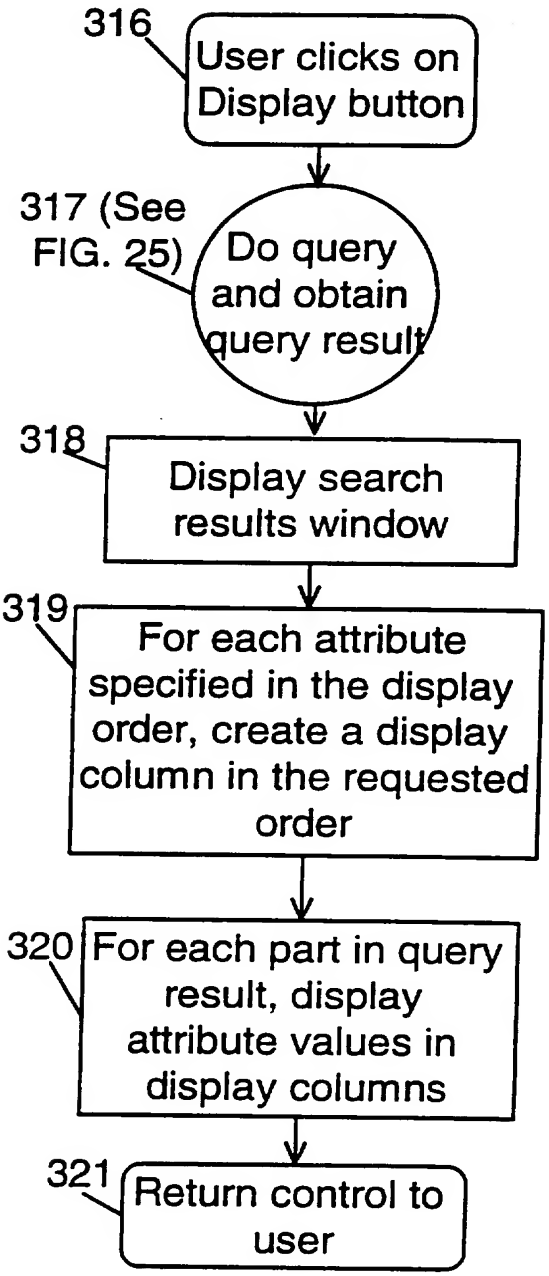

316
User clicks on Display button
317 (See FIG. 25)
Do query and obtain query result
318
Display search results window
319
For each attribute specified in the display order, create a display column in the requested order
320
For each part in query result, display attribute values in display columns
321
Return control to user

FIG. 23

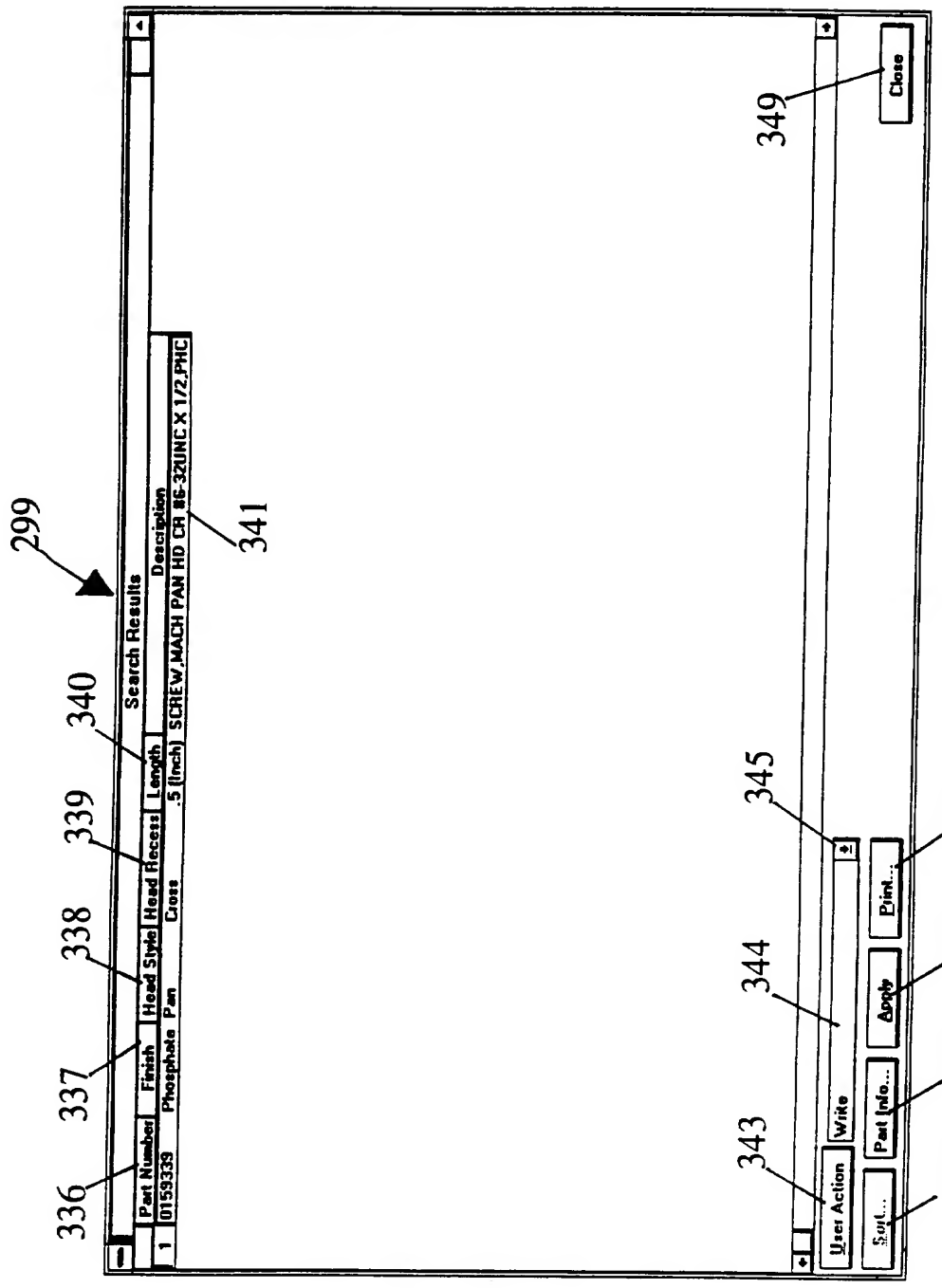

299
Search Results
Part Number
Finish
Head Style
Head Recess
Length
Description
1
0159339
Phosphate
Pan
Cross
.5 (inch)
SCREW,MACH PAN HD CR 86-32UNC X 1/2,PHC
336
337
338
339
340
341
349
Close
345
347
344
346
342
343
348
User Action
Write
Part Info...
Apply
Print...
Exit...

FIG. 24

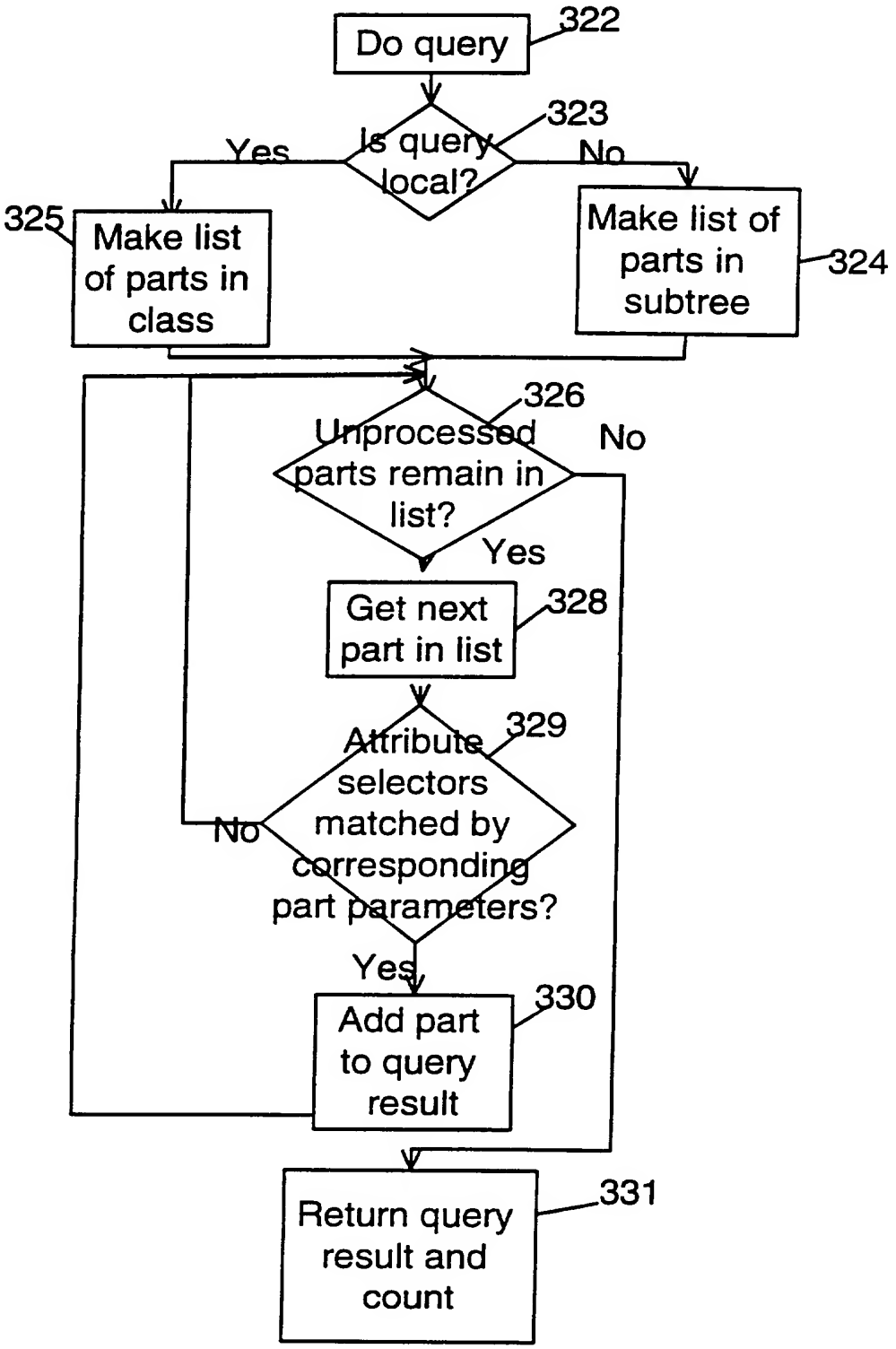

Do query
322
Is query local?
323
Yes
No
325
Make list of parts in class
Make list of parts in subtree
324
Unprocessed parts remain in list?
326
No
Yes
Get next part in list
328
Attribute selectors matched by corresponding part parameters?
329
No
Yes
Add part to query result
330
Return query result and count
331

FIG. 25

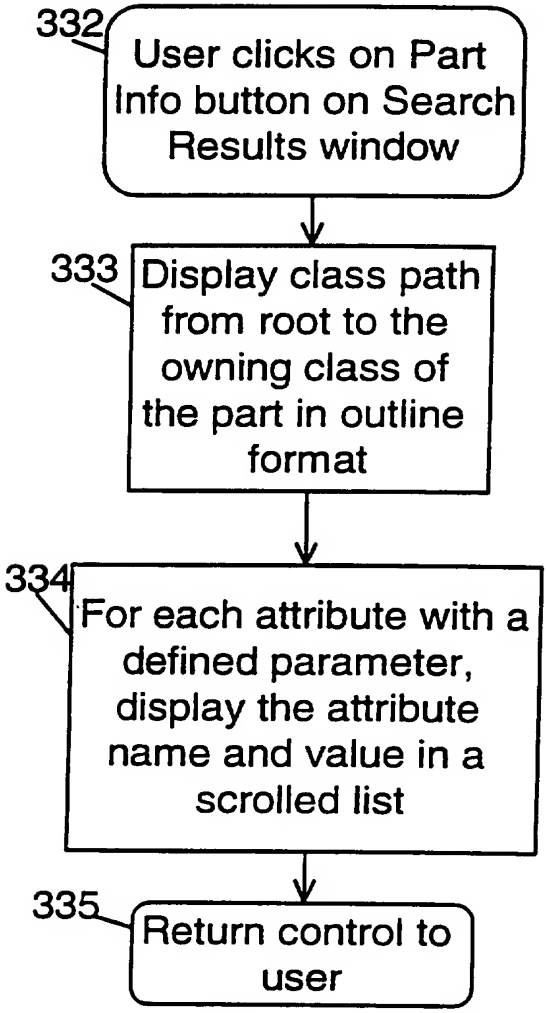

332
User clicks on Part Info button on Search Results window
333
Display class path from root to the owning class of the part in outline format
334
For each attribute with a defined parameter, display the attribute name and value in a scrolled list
335
Return control to user

FIG. 26

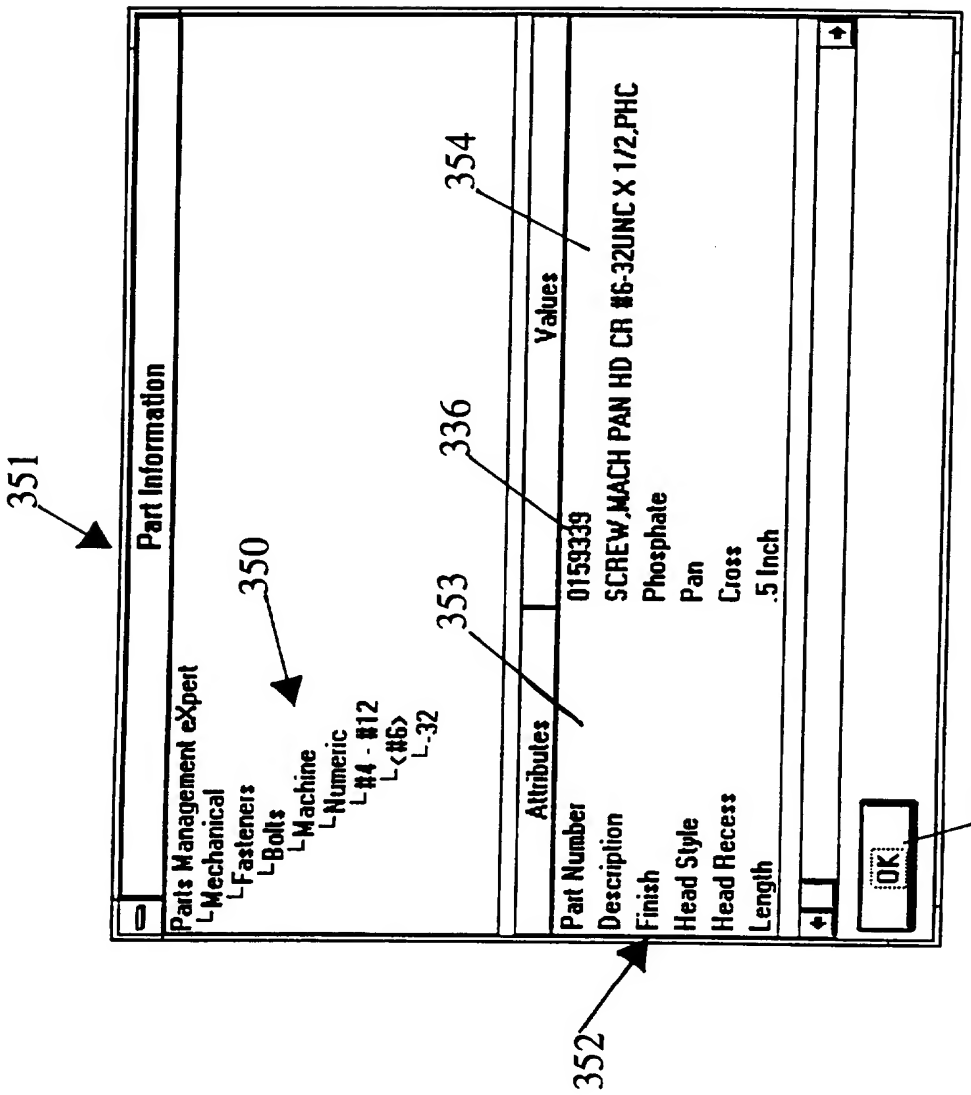

351
Part Information
Parts Management eXpert
Mechanical
Fasteners
Bolts
Machine
Numeric
#4 - #12
<#6>
-32
350
353
354
336
Attributes
Values
Part Number
Description
Finish
Head Style
Head Recess
Length
0159339
SCREW,MACH PAN HD CR #6-32UNC X 1/2,PHC
Phosphate
Pan
Cross
.5 Inch
OK
356
352

FIG. 27

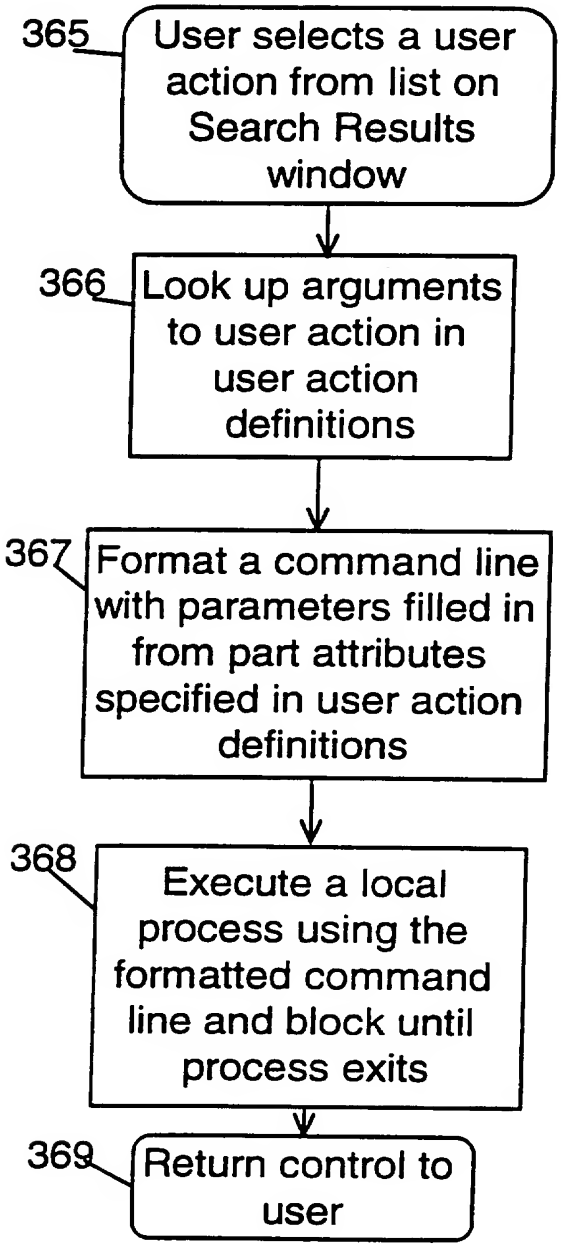

365
User selects a user action from list on Search Results window
366
Look up arguments to user action in user action definitions
367
Format a command line with parameters filled in from part attributes specified in user action definitions
368
Execute a local process using the formatted command line and block until process exits
369
Return control to user

FIG. 28

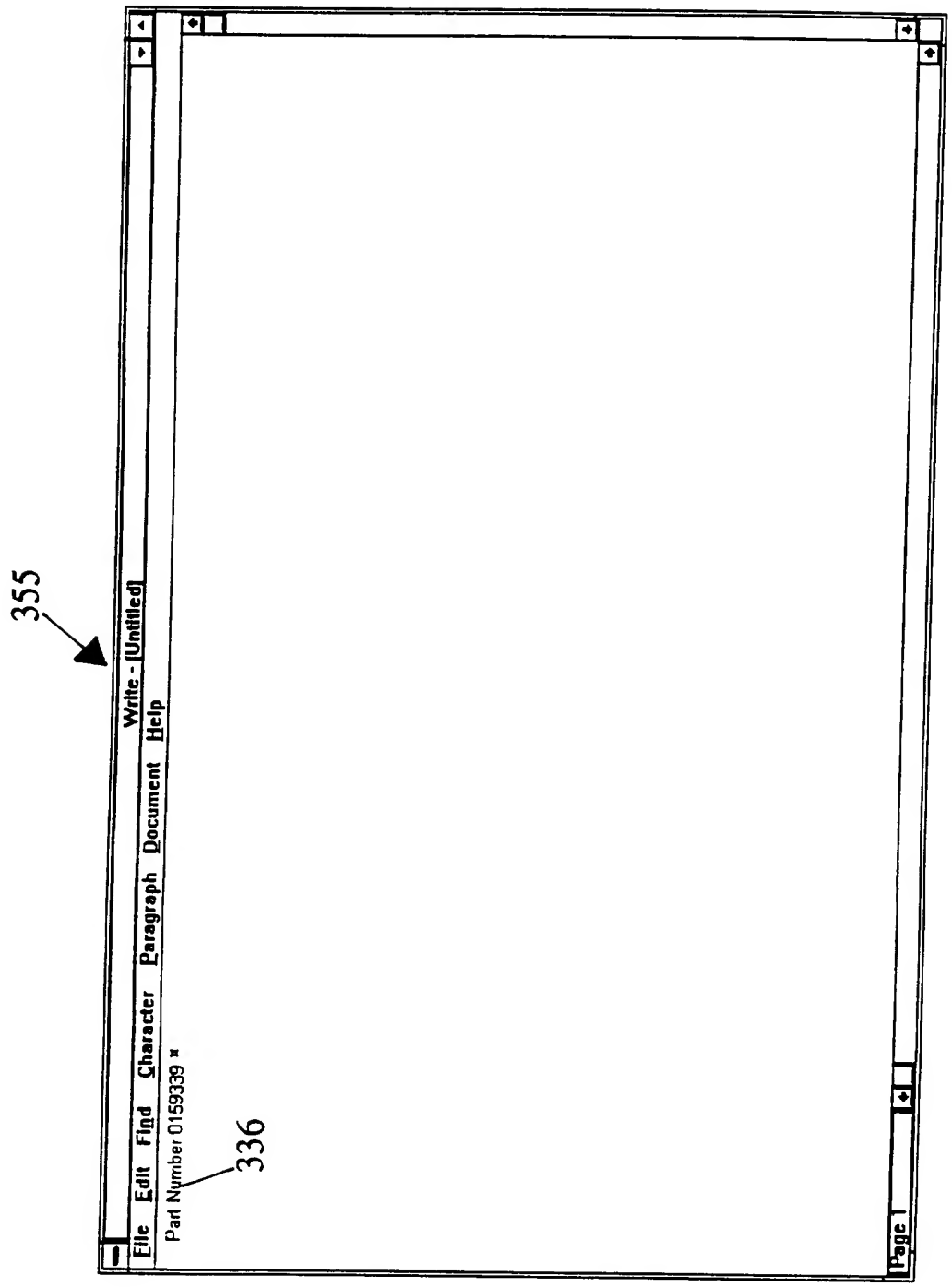

355
Write - [Untitled]
File Edit Find Character Paragraph Document Help
Part Number 0159339
336
Page 1

FIG. 29

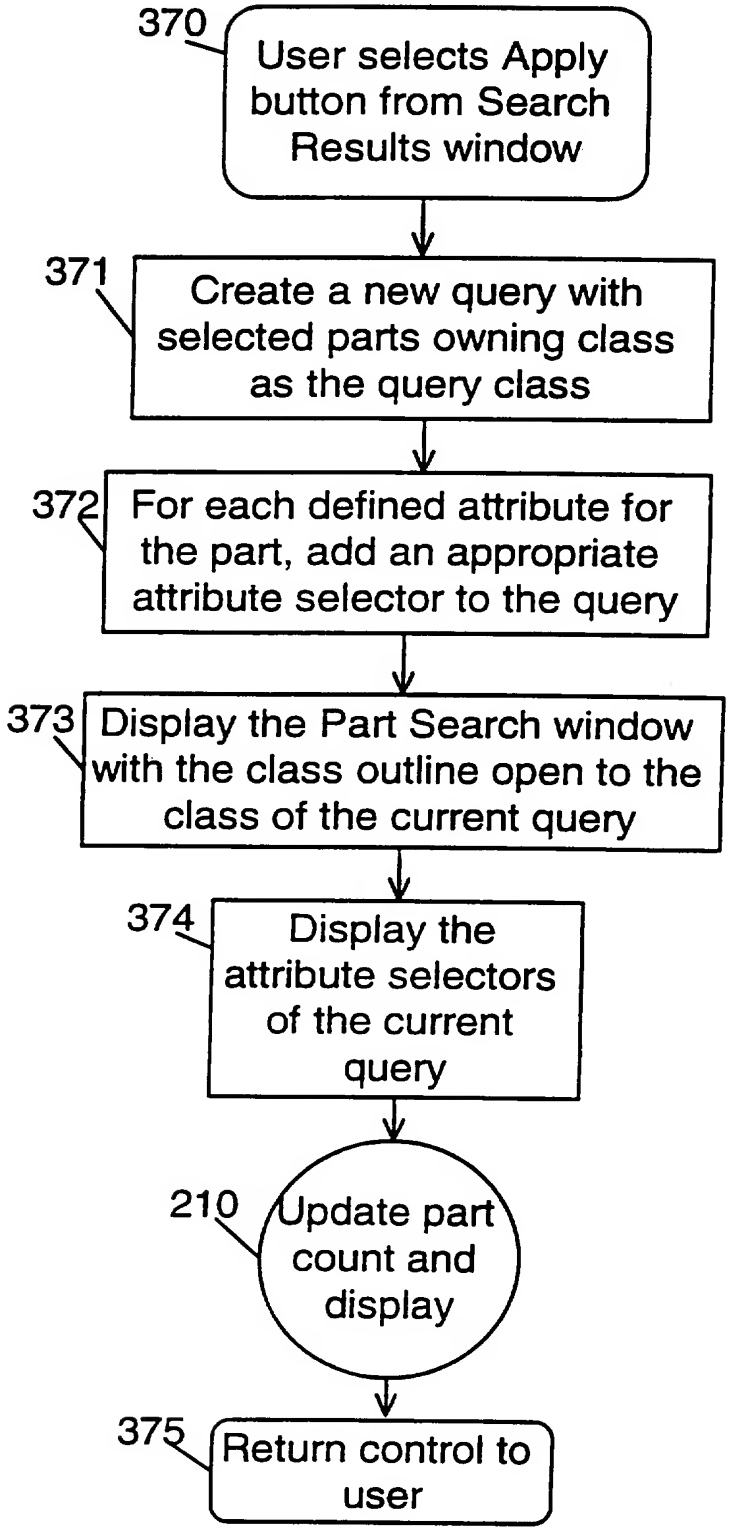

370
User selects Apply button from Search Results window
371
Create a new query with selected parts owning class as the query class
372
For each defined attribute for the part, add an appropriate attribute selector to the query
373
Display the Part Search window with the class outline open to the class of the current query
374
Display the attribute selectors of the current query
210
Update part count and display
375
Return control to user

FIG. 30

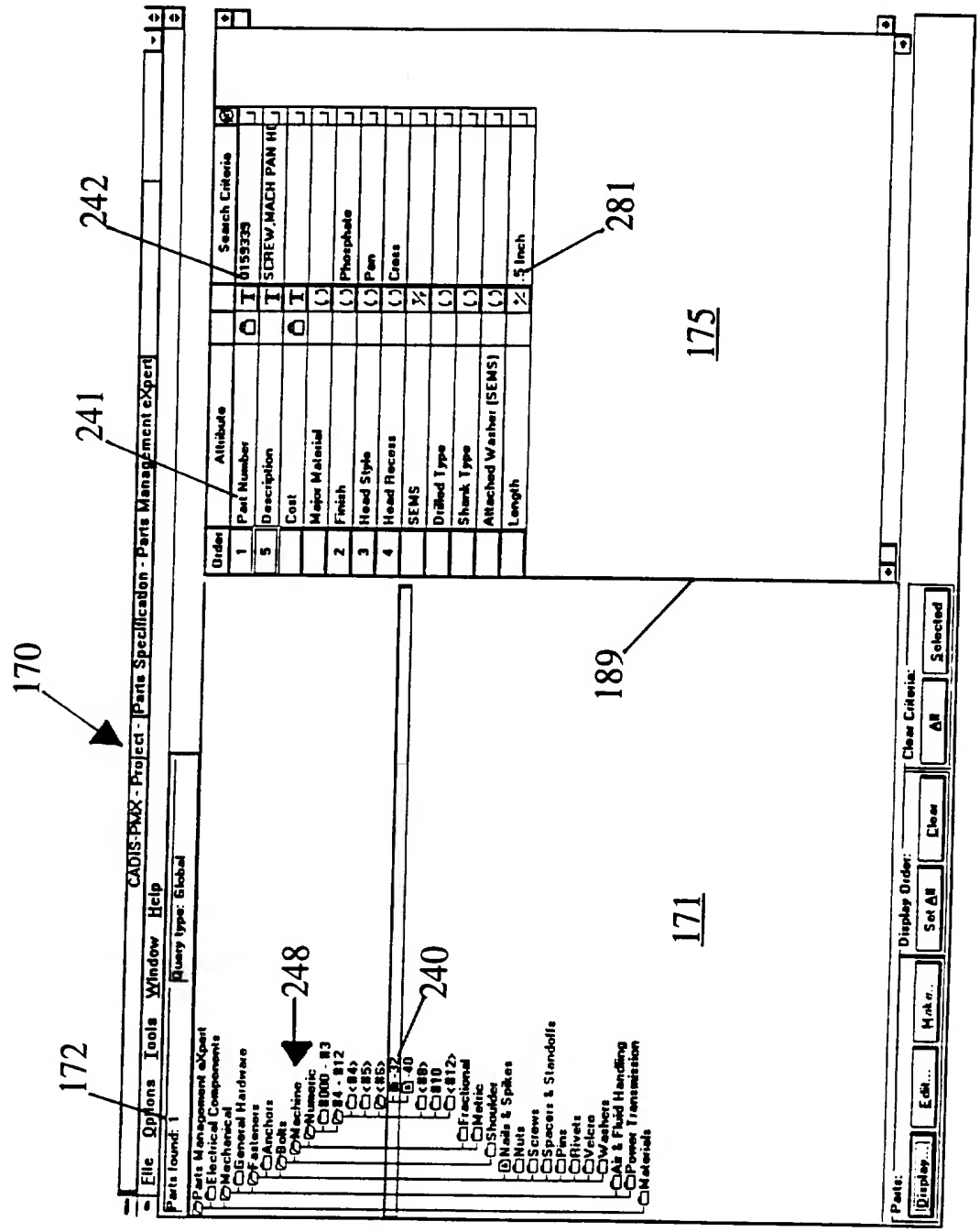

242
241
281
175
170
189
172
248
240
171
CADIS-PMX - Project - Parts Specification - Parts Management eXpert
File Options Tools Window Help
Parts found: 1
Query type: Global
Search Criteria
Order
Attribute
Part Number
Description
Cost
Major Material
Finish
Head Style
Head Recess
SEMS
Drilled Type
Shank Type
Attached Washer (SEMS)
Length
0159339
SCREW,MACH PAN HD
Phosphate
Pan
Cross
5 Inch
Parts Management eXpert
Electrical Components
Mechanical Components
General Hardware
Fasteners
Anchors
Bolts
Machine
Numeric
Fractional
Metric
Shoulder
Nails & Spikes
Nuts
Screws
Spacers & Standoffs
Pins
Rivets
Velcro
Washers
Air & Fluid Handling
Power Transmission
Materials
Parts:
Display...
Edit...
Display Order:
Set All
Clear
Clear Criteria:
All
Selected

FIG. 31

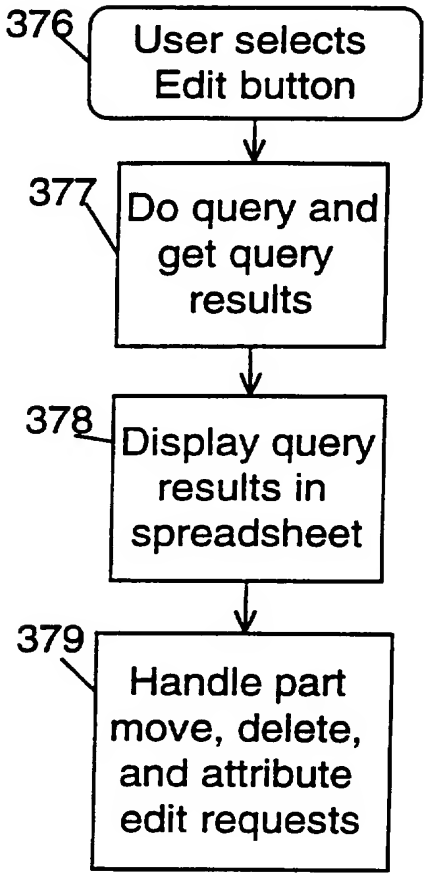

376
User selects Edit button
377
Do query and get query results
378
Display query results in spreadsheet
379
Handle part move, delete, and attribute edit requests

FIG. 32

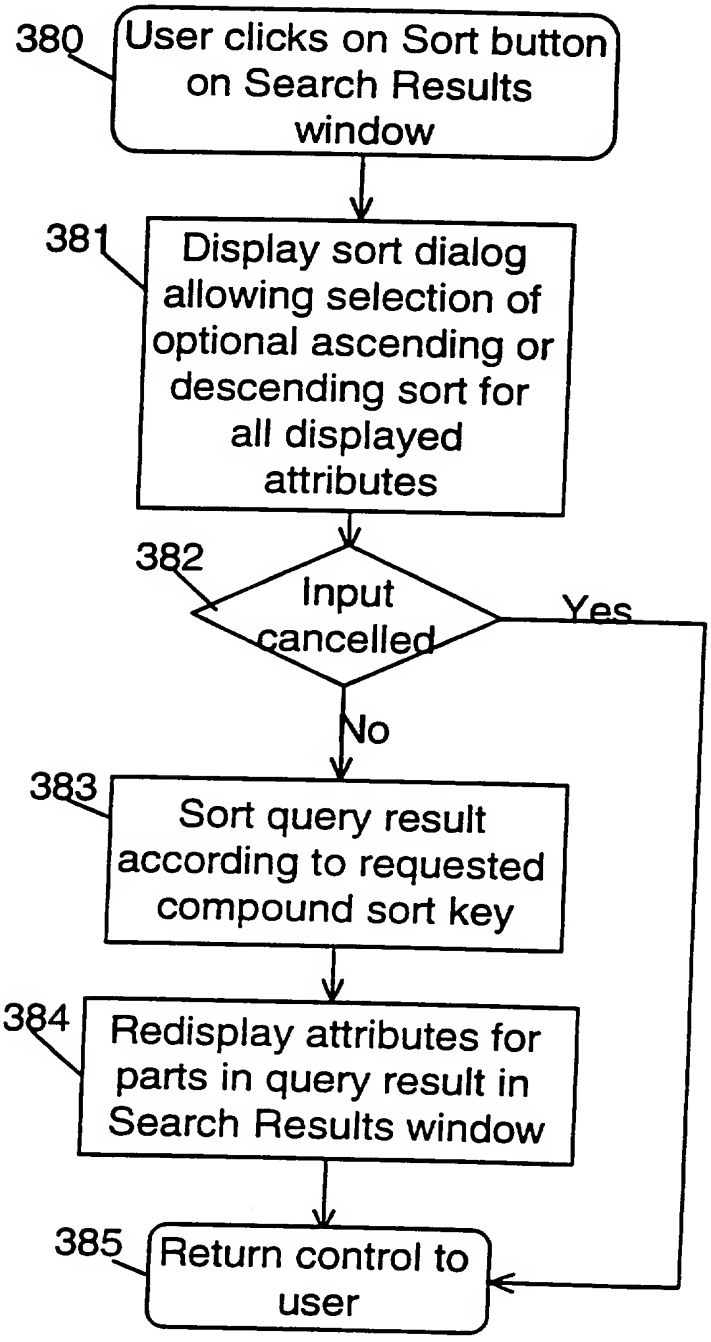

380
User clicks on Sort button on Search Results window
381
Display sort dialog allowing selection of optional ascending or descending sort for all displayed attributes
382
Input cancelled
Yes
No
383
Sort query result according to requested compound sort key
384
Redisplay attributes for parts in query result in Search Results window
385
Return control to user

FIG. 33

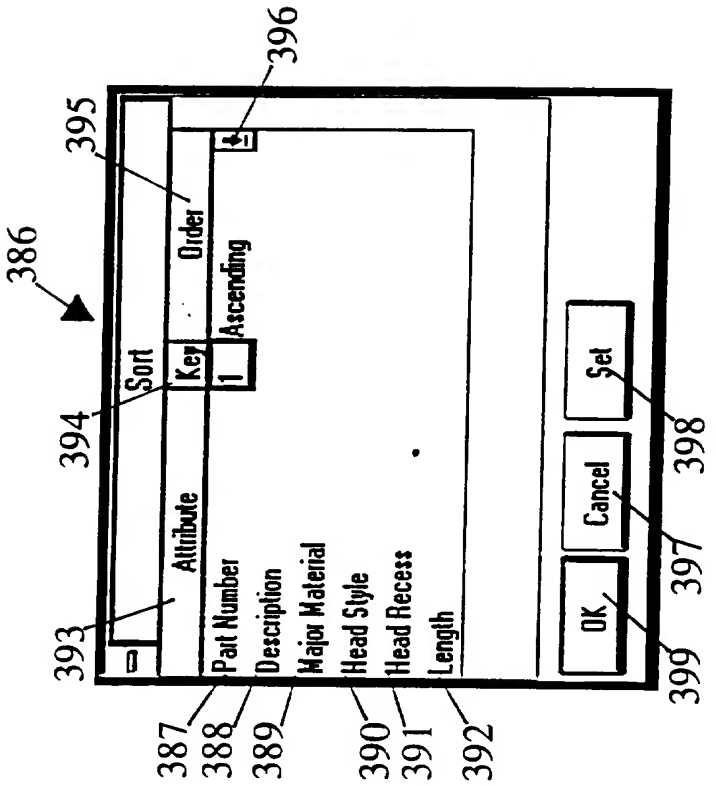

386
393
394
395
396
Sort
Key
1
Order
Ascending
Attribute
Part Number
Description
Major Material
Head Style
Head Recess
Length
OK
Cancel
Set
387
388
389
390
391
392
397
398
399

FIG. 34

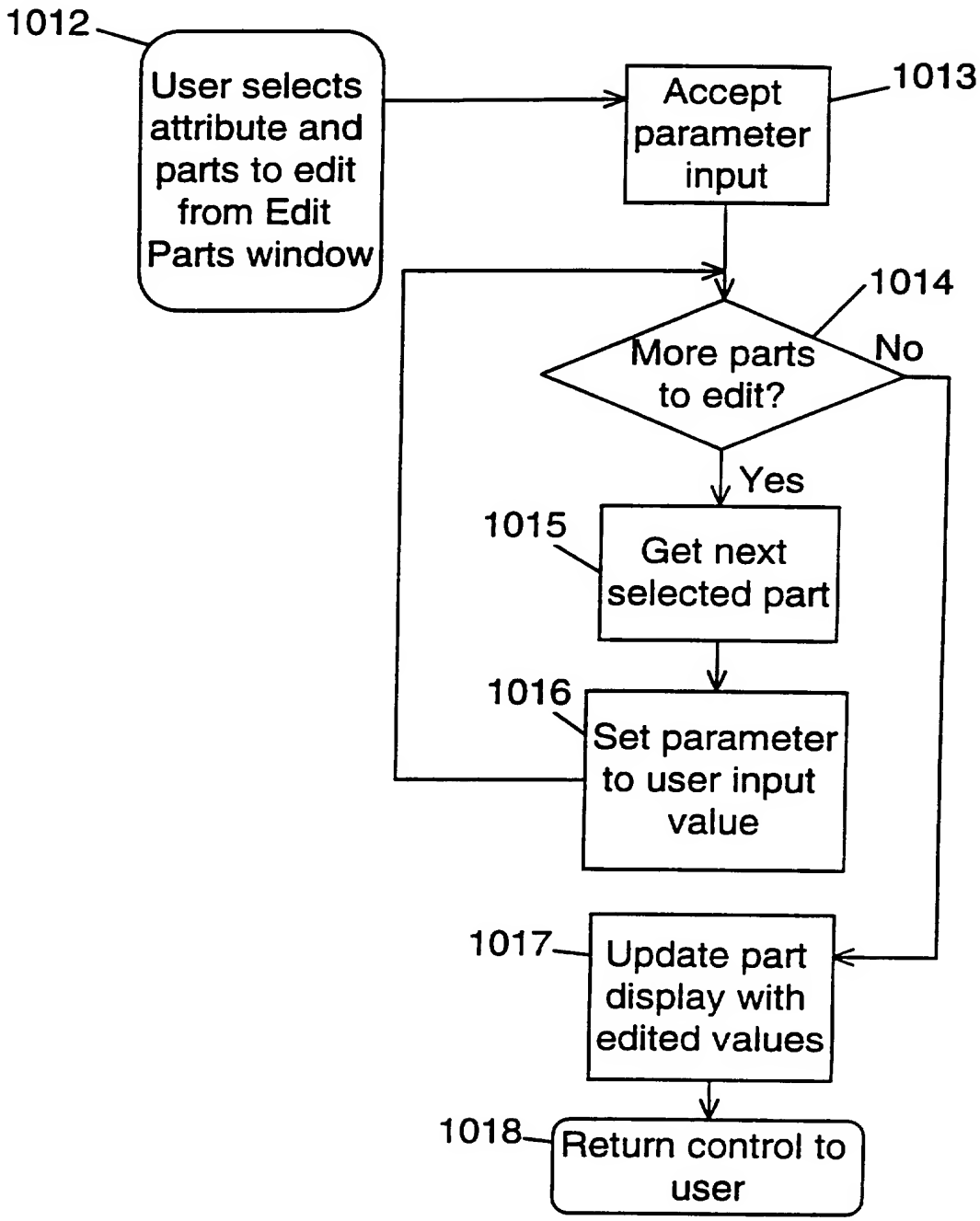

1012
User selects attribute and parts to edit from Edit Parts window
Accept parameter input
1013
1014
More parts to edit?
No
Yes
1015
Get next selected part
1016
Set parameter to user input value
1017
Update part display with edited values
1018
Return control to user

FIG. 35

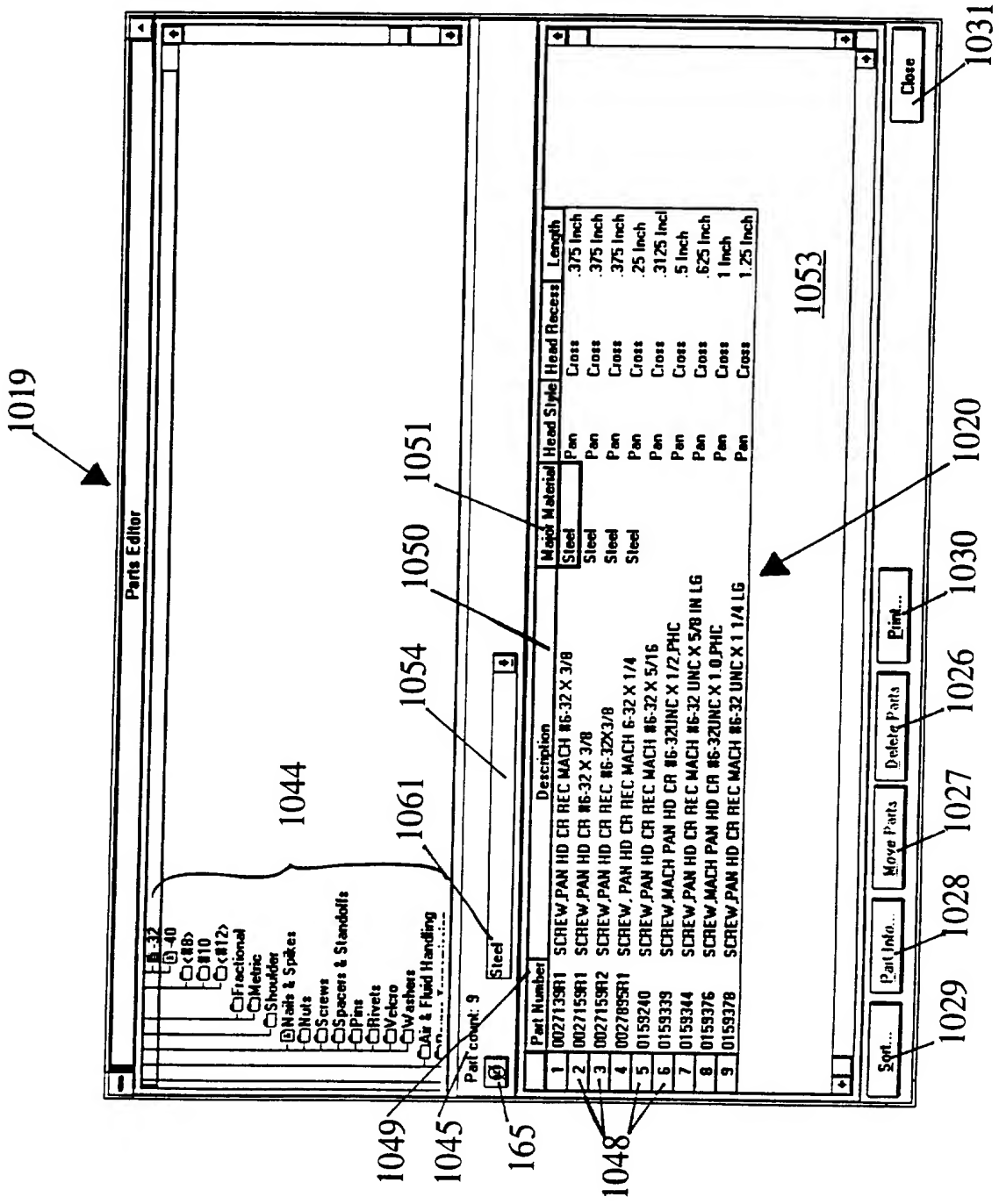

1019
1044
1061
1054
1050
1051
1053
1020
1031
1030
1026
1027
1028
1029
1049
1045
165
1048
Parts Editor
Close
Print...
Delete Parts
Move Parts
Part Info...
Sort...
Part count: 9
Steel
Part Number
Description
Major Material
Head Style
Head Recess
Length
1 0027139R1 SCREW,PAN HD CR REC MACH #6-32 X 3/8 Steel Pan Cross .375 Inch
2 0027159R1 SCREW,PAN HD CR #6-32 X 3/8 Steel Pan Cross .375 Inch
3 0027159R2 SCREW,PAN HD CR REC #6-32X3/8 Steel Pan Cross .375 Inch
4 0027895R1 SCREW, PAN HD CR REC MACH 6-32 X 1/4 Steel Pan Cross .25 Inch
5 0159240 SCREW,PAN HD CR REC MACH #6-32 X 5/16 Pan Cross .3125 Incl
6 0159339 SCREW,MACH PAN HD CR #6-32UNC X 1/2,PHC Pan Cross .5 Inch
7 0159344 SCREW,PAN HD CR REC MACH #6-32 UNC X 5/8 IN LG Pan Cross .625 Inch
8 0159376 SCREW,MACH PAN HD CR #6-32UNC X 1.0,PHC Pan Cross 1 Inch
9 0159378 SCREW,PAN HD CR REC MACH #6-32 UNC X 1 1/4 LG Pan Cross 1.25 Inch

FIG. 36

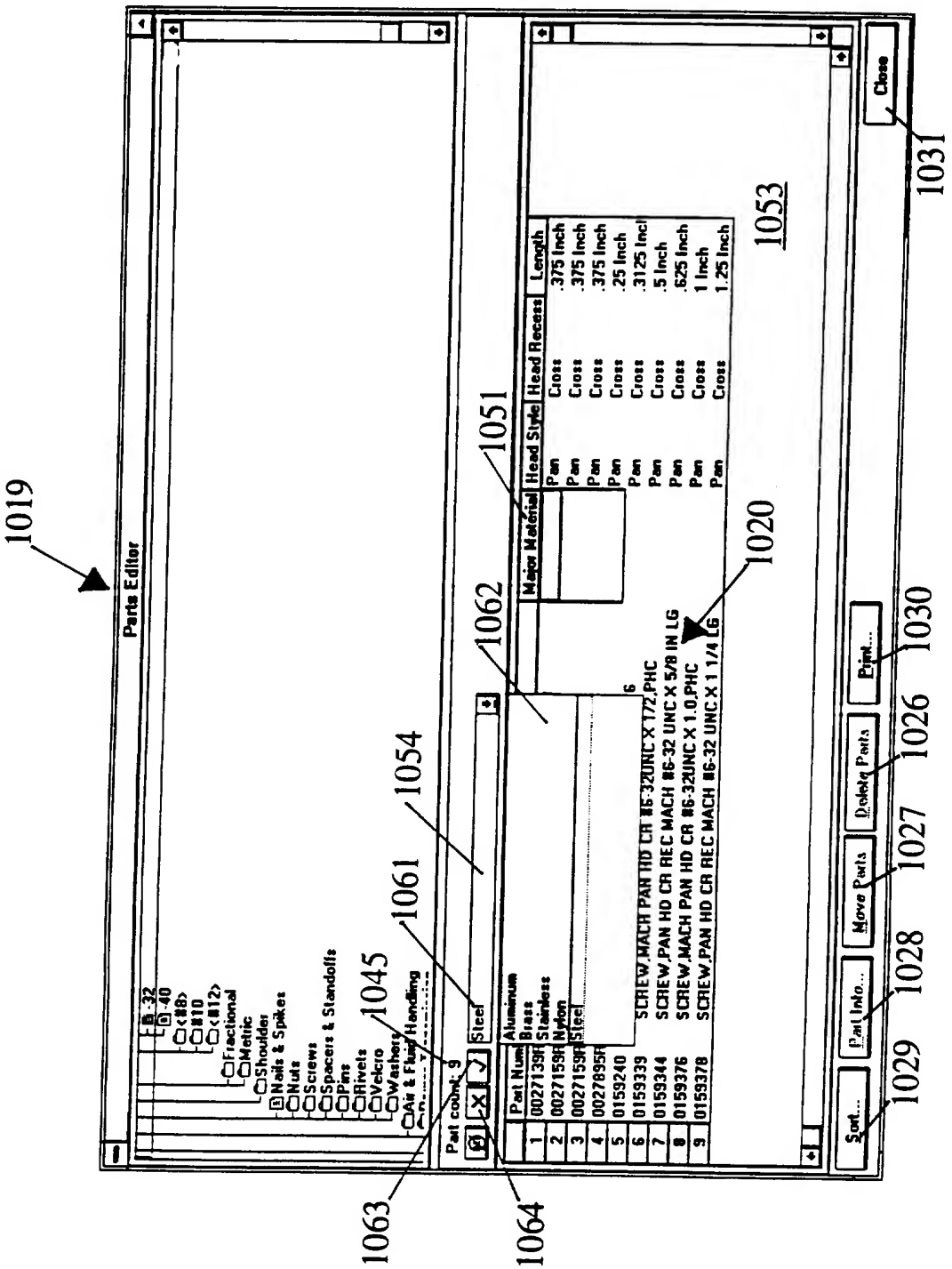

1019
Parts Editor
B-32
B-40
<B8>
B10
<B12>
Fractional
Metric
Shoulder
Nails & Spikes
Nuts
Screws
Spacers & Standoffs
Pins
Rivets
Velcro
Washers
Air & Fluid Handling
1045
1061
1063
Part count: 9
Steel
1064
1054
1062
1051
1053
1020
Close
1031
Part Num
Aluminum
Brass
Stainless
Nylon
Steel
Major Material
Head Style
Head Recess
Length
1 0027139R Pan Cross .375 Inch
2 0027159R Pan Cross .375 Inch
3 0027159R Pan Cross .375 Inch
4 0027895R Pan Cross .25 Inch
5 0159240 Pan Cross .3125 Inch
6 0159339 SCREW,MACH PAN HD CR 86-32UNC X 172,PHC Pan Cross .5 Inch
7 0159344 SCREW,PAN HD CR REC MACH 86-32 UNC X 5/8 IN LG Pan Cross .625 Inch
8 0159376 SCREW,MACH PAN HD CR 86-32UNC X 1.0,PHC Pan Cross 1 Inch
9 0159378 SCREW,PAN HD CR REC MACH 86-32 UNC X 1 1/4 LG Pan Cross 1.25 Inch
1030
Print...
1026
Delete Parts
1027
Move Parts
1028
Part Info...
1029
Sort...

FIG. 37

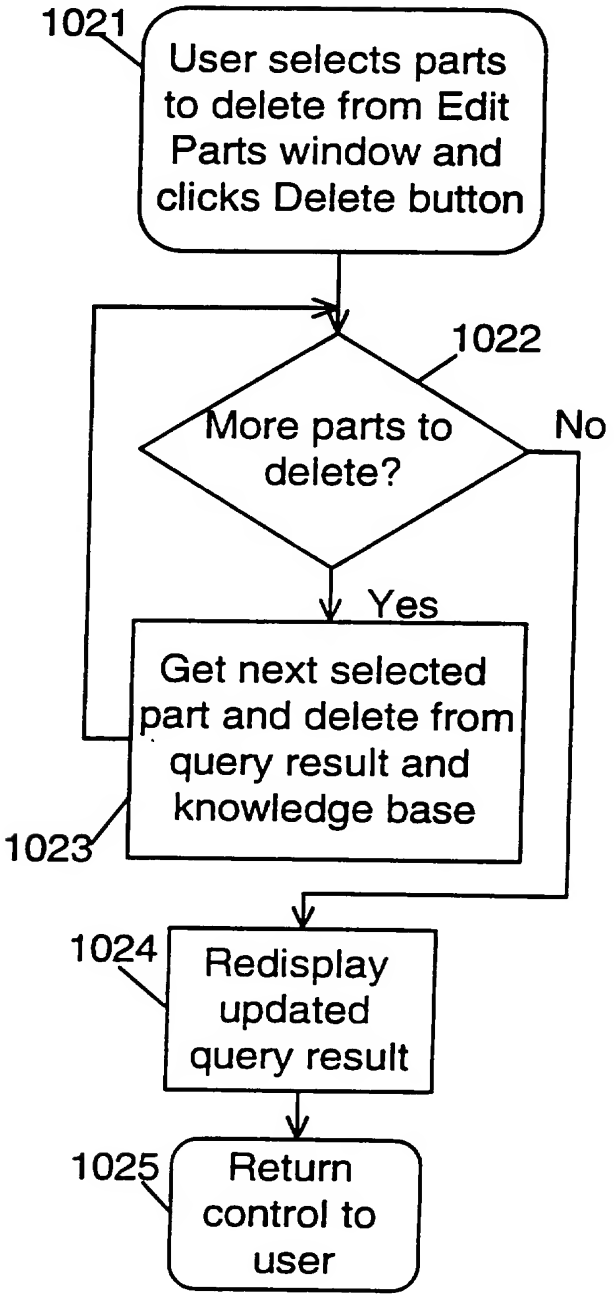

1021
User selects parts to delete from Edit Parts window and clicks Delete button
1022
More parts to delete?
No
Yes
Get next selected part and delete from query result and knowledge base
1023
1024
Redisplay updated query result
1025
Return control to user

FIG. 38

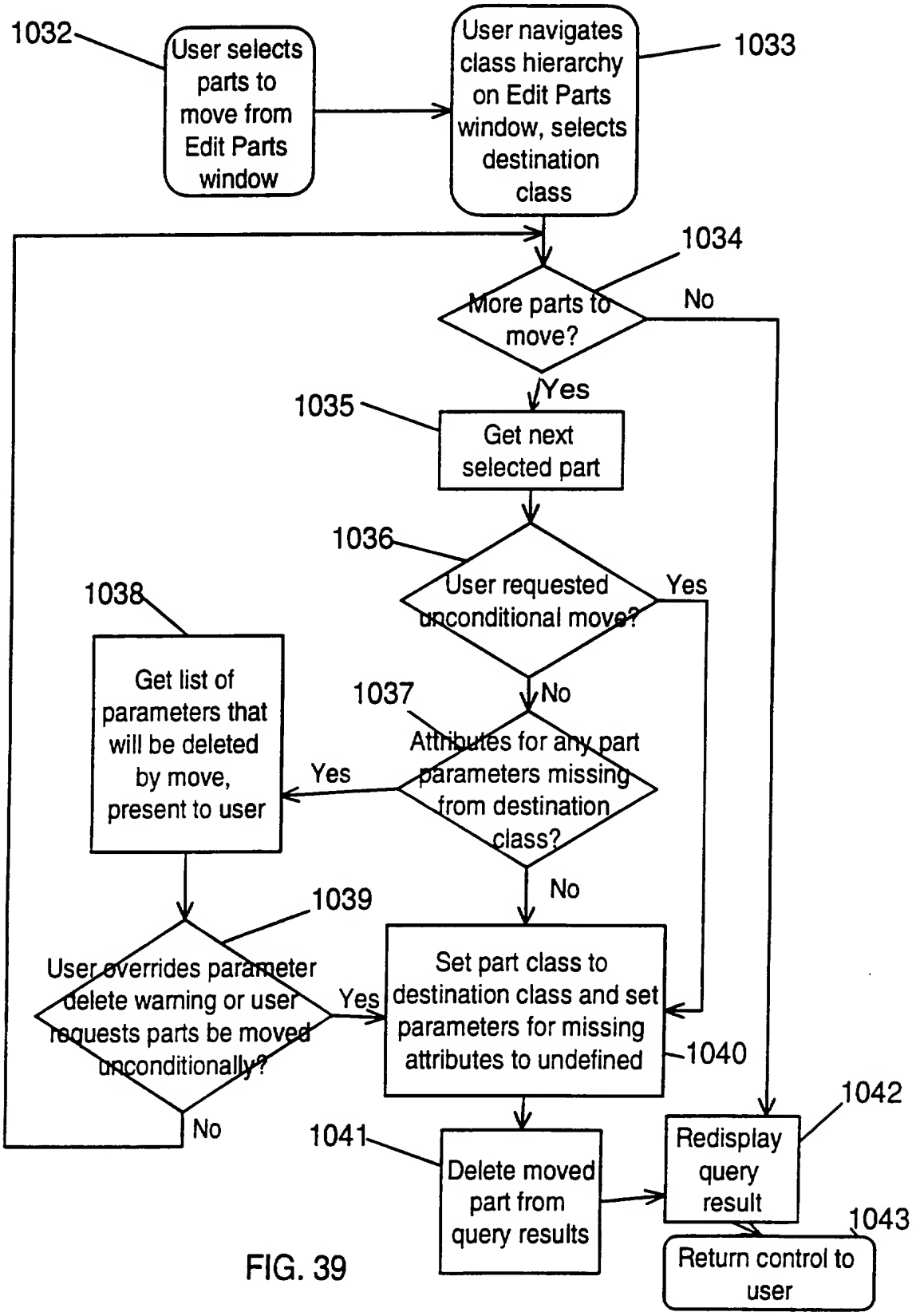

1032
User selects parts to move from Edit Parts window
User navigates class hierarchy on Edit Parts window, selects destination class
1033
1034
More parts to move?
No
Yes
1035
Get next selected part
1036
User requested unconditional move?
Yes
1038
Get list of parameters that will be deleted by move, present to user
1037
No
Attributes for any part parameters missing from destination class?
Yes
No
1039
User overrides parameter delete warning or user requests parts be moved unconditionally?
Yes
No
Set part class to destination class and set parameters for missing attributes to undefined
1040
1042
1041
Delete moved part from query results
Redisplay query result
1043
Return control to user

FIG. 39

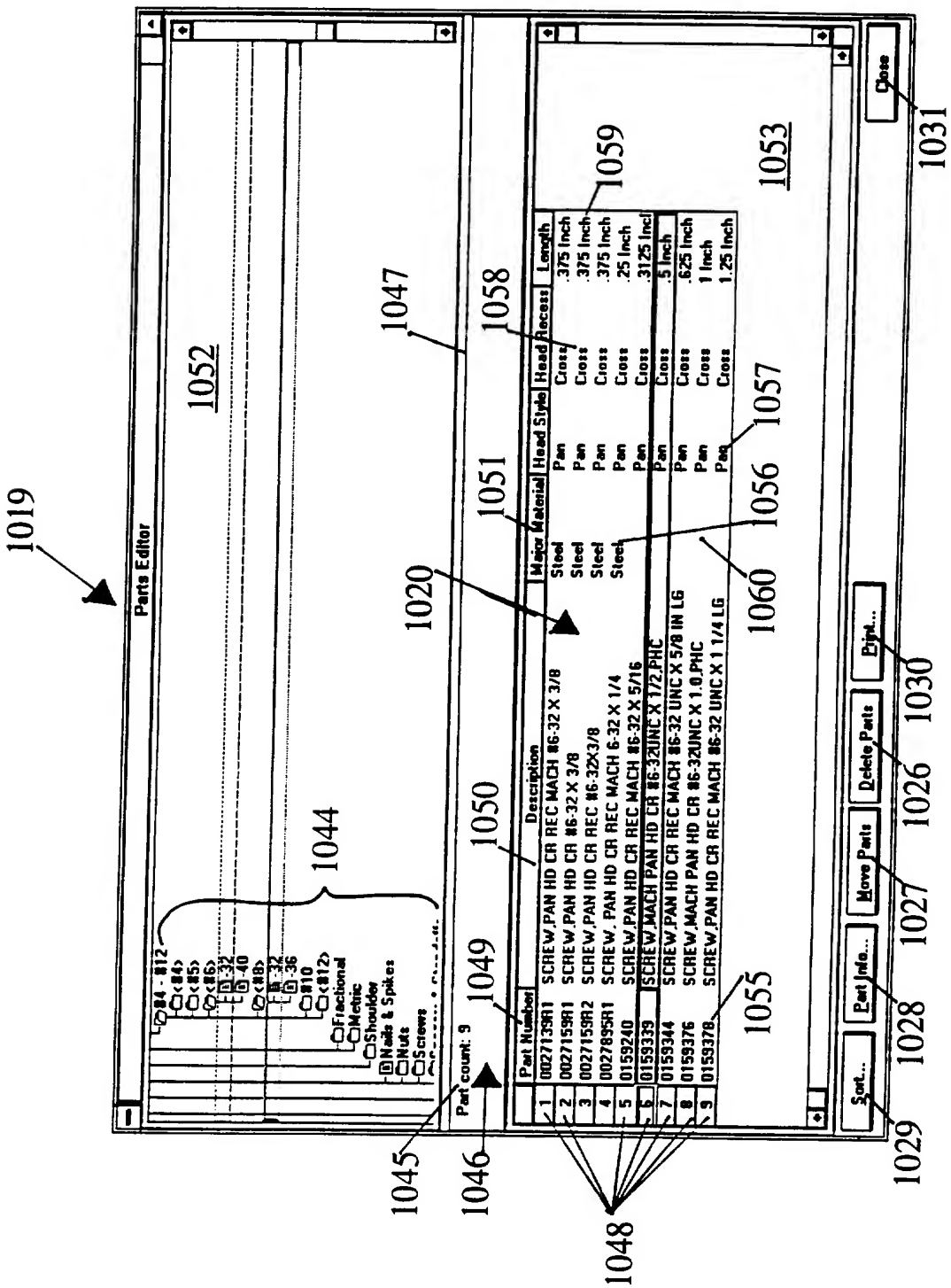

Parts Editor
Fractional
Metric
Shoulder
Nails & Spikes
Nuts
Screws
Part count: 9
Part Number
Description
Major Material
Head Style
Head Recess
Length
Sort...
Part Info...
Move Parts
Delete Parts
Print...
Close
1019
1020
1027
1028
1029
1030
1031
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1055
1056
1057
1058
1059
1060

FIG. 40

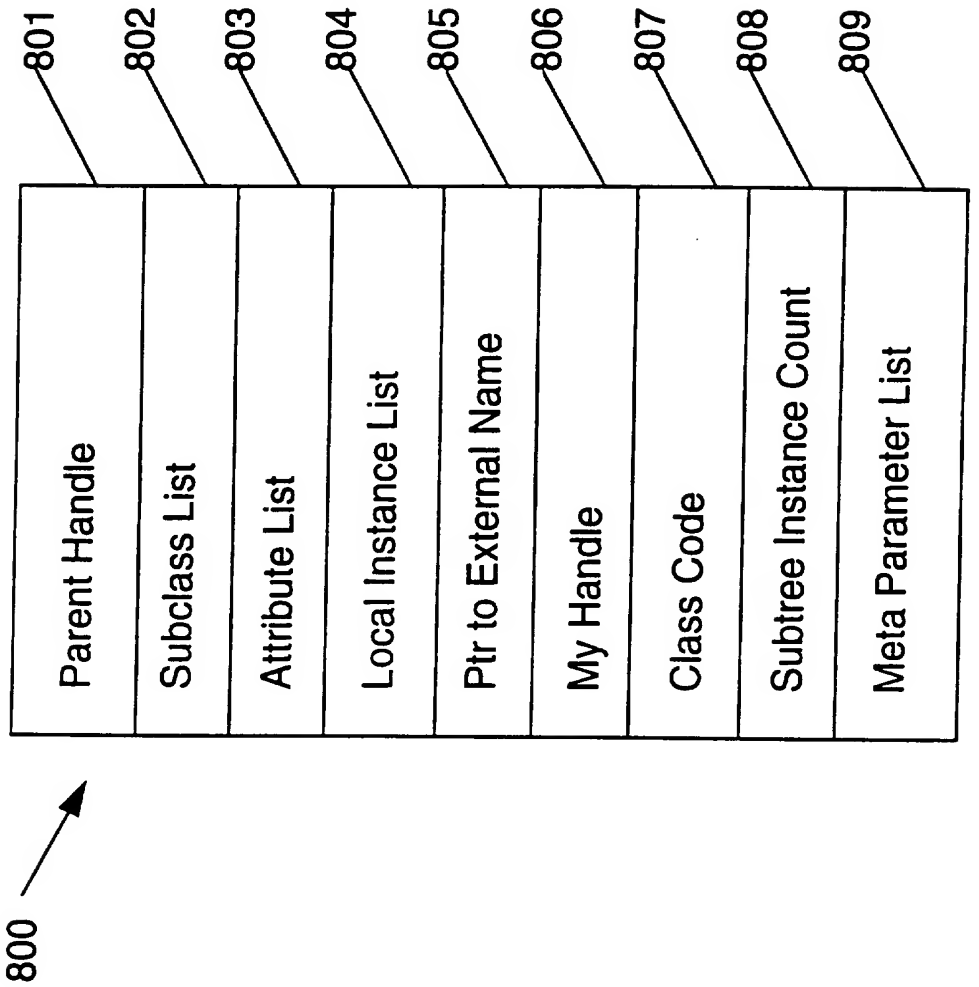

801
802
803
804
805
806
807
808
809
Parent Handle
Subclass List
Attribute List
Local Instance List
Ptr to External Name
My Handle
Class Code
Subtree Instance Count
Meta Parameter List
800

FIG. 41

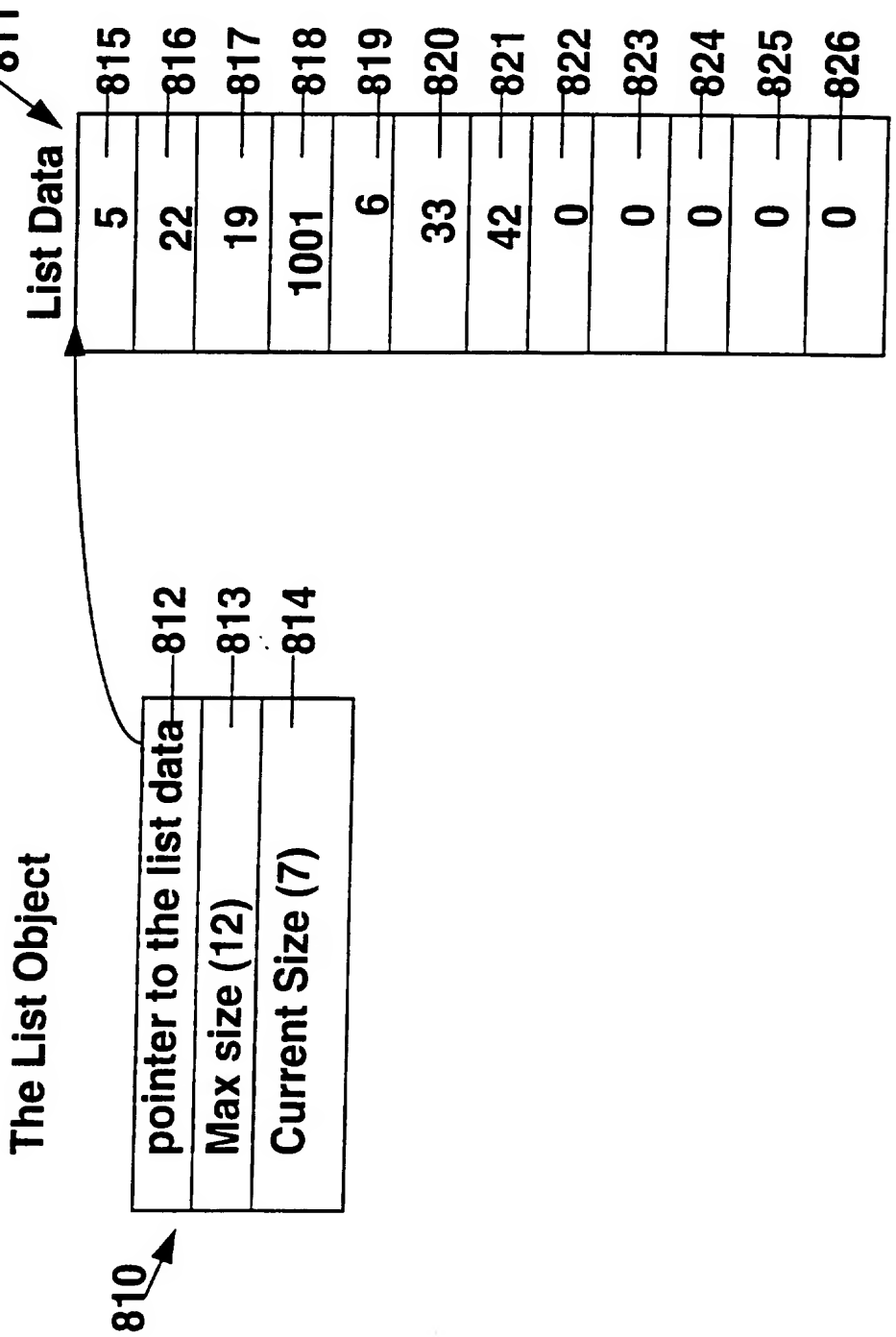

811
List Data
815
816
817
818
819
820
821
822
823
824
825
826
5
22
19
1001
6
33
42
0
0
0
0
0
812
813
814
The List Object
pointer to the list data
Max size (12)
Current Size (7)
810

FIG. 42

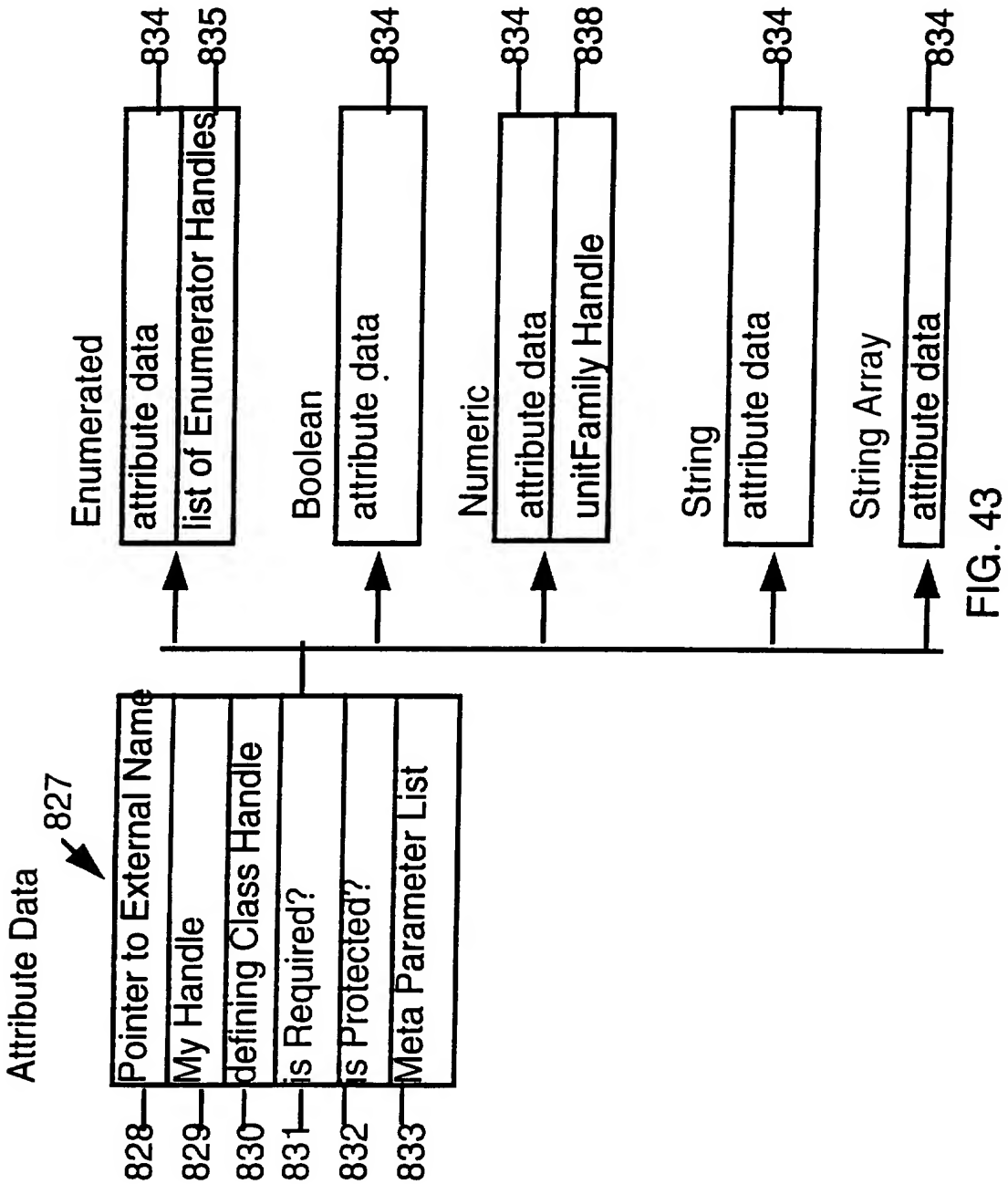

Attribute Data
827
Pointer to External Name
My Handle
defining Class Handle
is Required?
is Protected?
Meta Parameter List
828
829
830
831
832
833
Enumerated
attribute data
list of Enumerator Handles
834
835
Boolean
attribute data
834
Numeric
attribute data
unitFamily Handle
834
838
String
attribute data
834
String Array
attribute data
834

FIG. 43

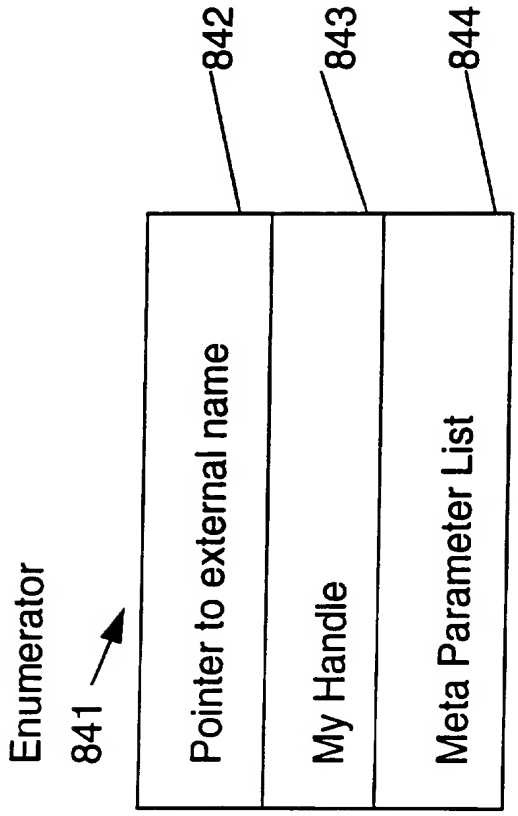

842
843
844
Enumerator
841
Pointer to external name
My Handle
Meta Parameter List

FIG. 44

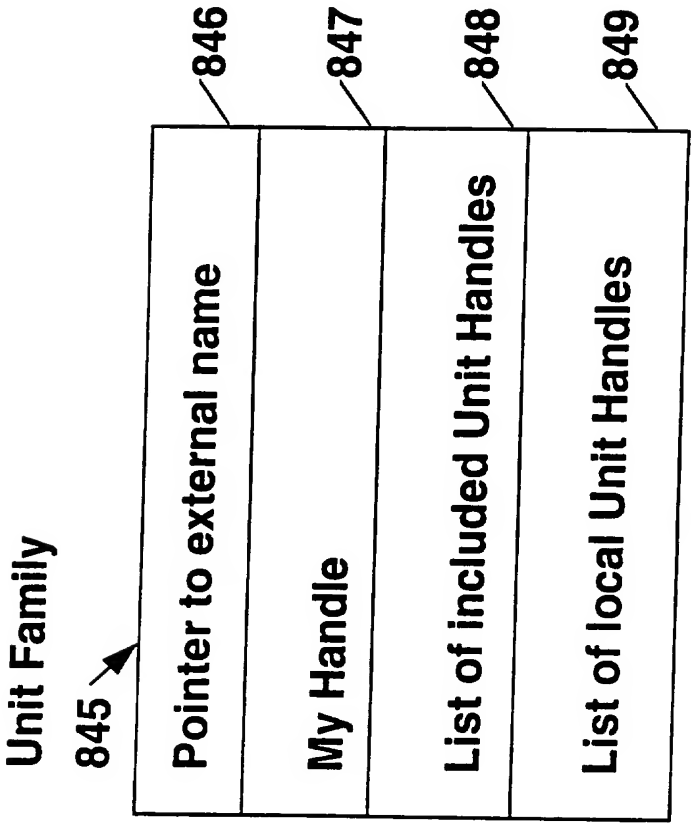

Unit Family
845
Pointer to external name
My Handle
List of included Unit Handles
List of local Unit Handles
846
847
848
849

FIG. 45

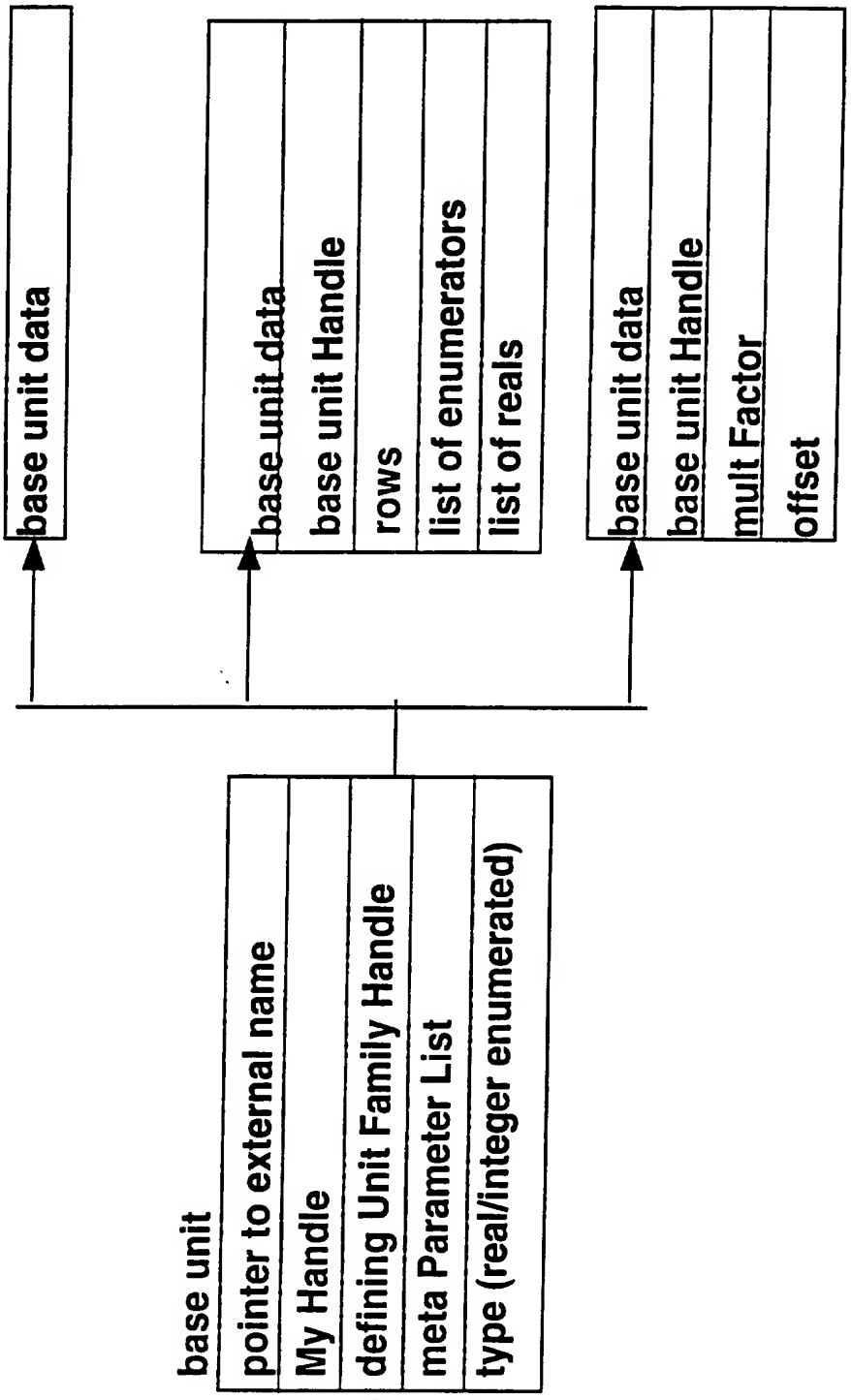

base unit data
base unit data
base unit Handle
rows
list of enumerators
list of reals
base unit data
base unit Handle
mult Factor
offset
base unit
pointer to external name
My Handle
defining Unit Family Handle
meta Parameter List
type (real/integer enumerated)

FIG. 46

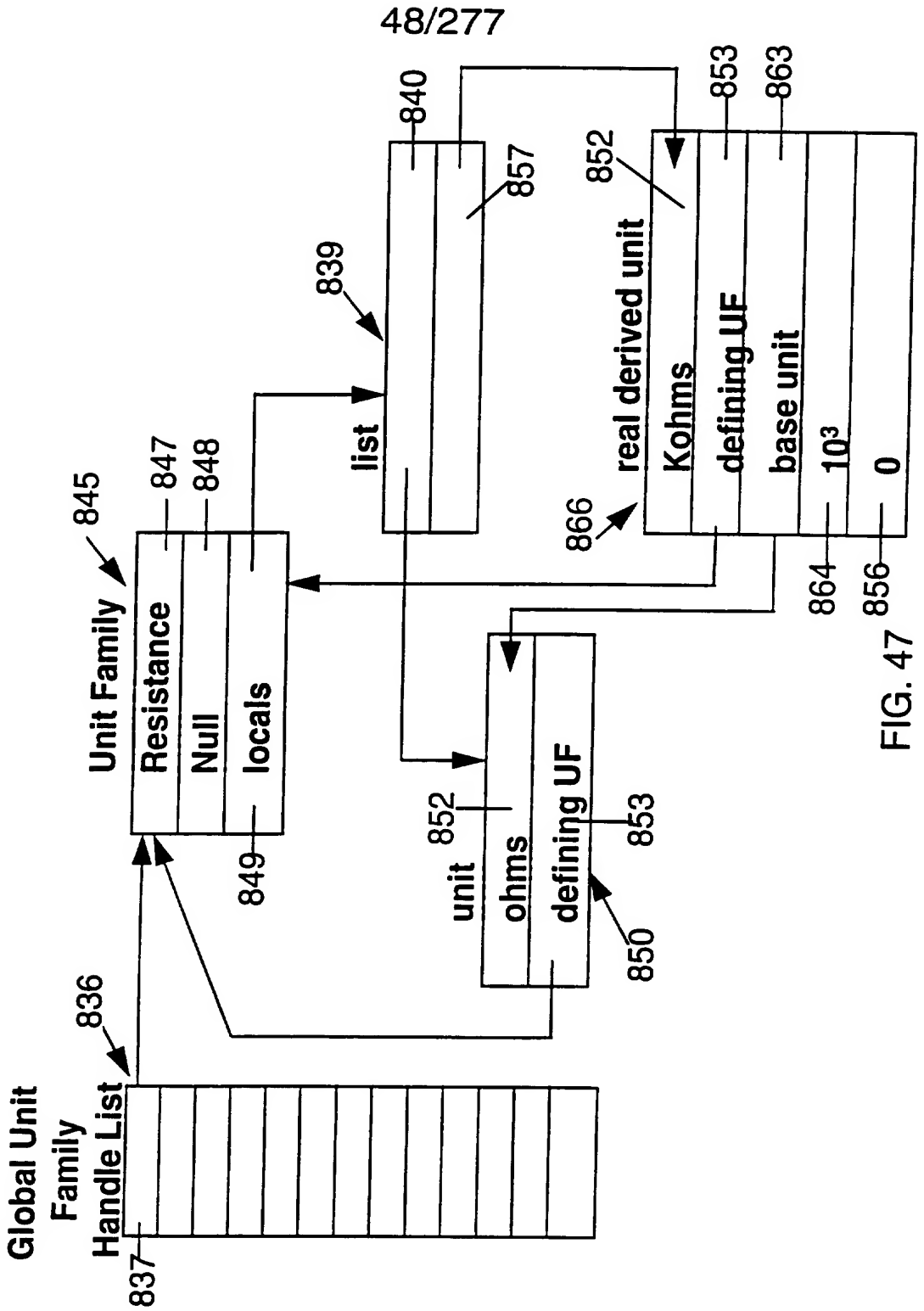

Global Unit Family Handle List
836
837
Unit Family
845
Resistance
Null
locals
847
848
849
list
839
840
857
unit
ohms
defining UF
852
853
850
real derived unit
Kohms
defining UF
base unit
10^3
0
852
853
863
866
864
856

FIG. 47

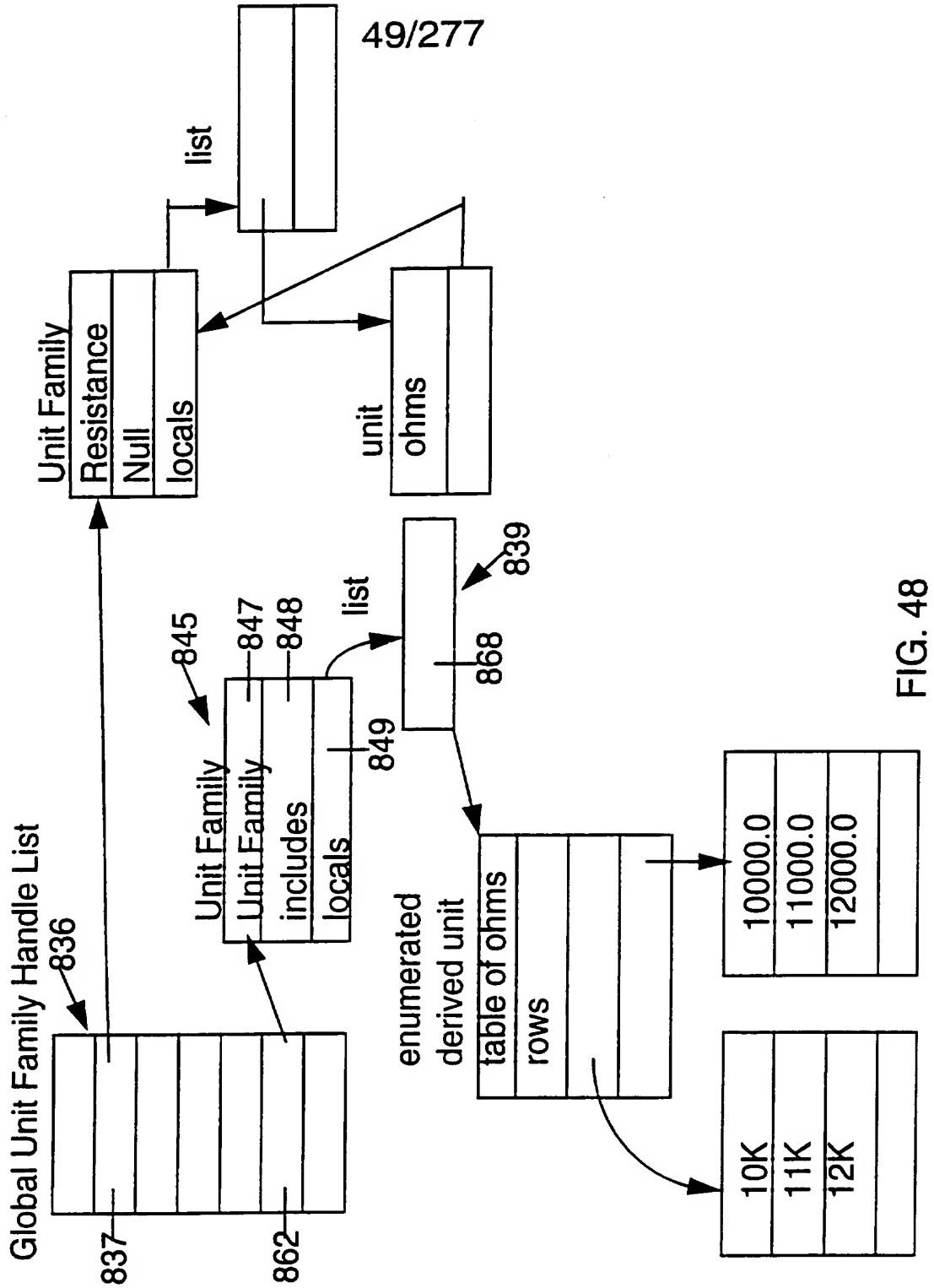

49/277
list
Unit Family
Resistance
Null
locals
unit
ohms
839
845
847
848
list
868
849
Unit Family
Unit Family
includes
locals
Global Unit Family Handle List
836
enumerated
derived unit
table of ohms
rows
10000.0
11000.0
12000.0
10K
11K
12K
837
862

FIG. 48

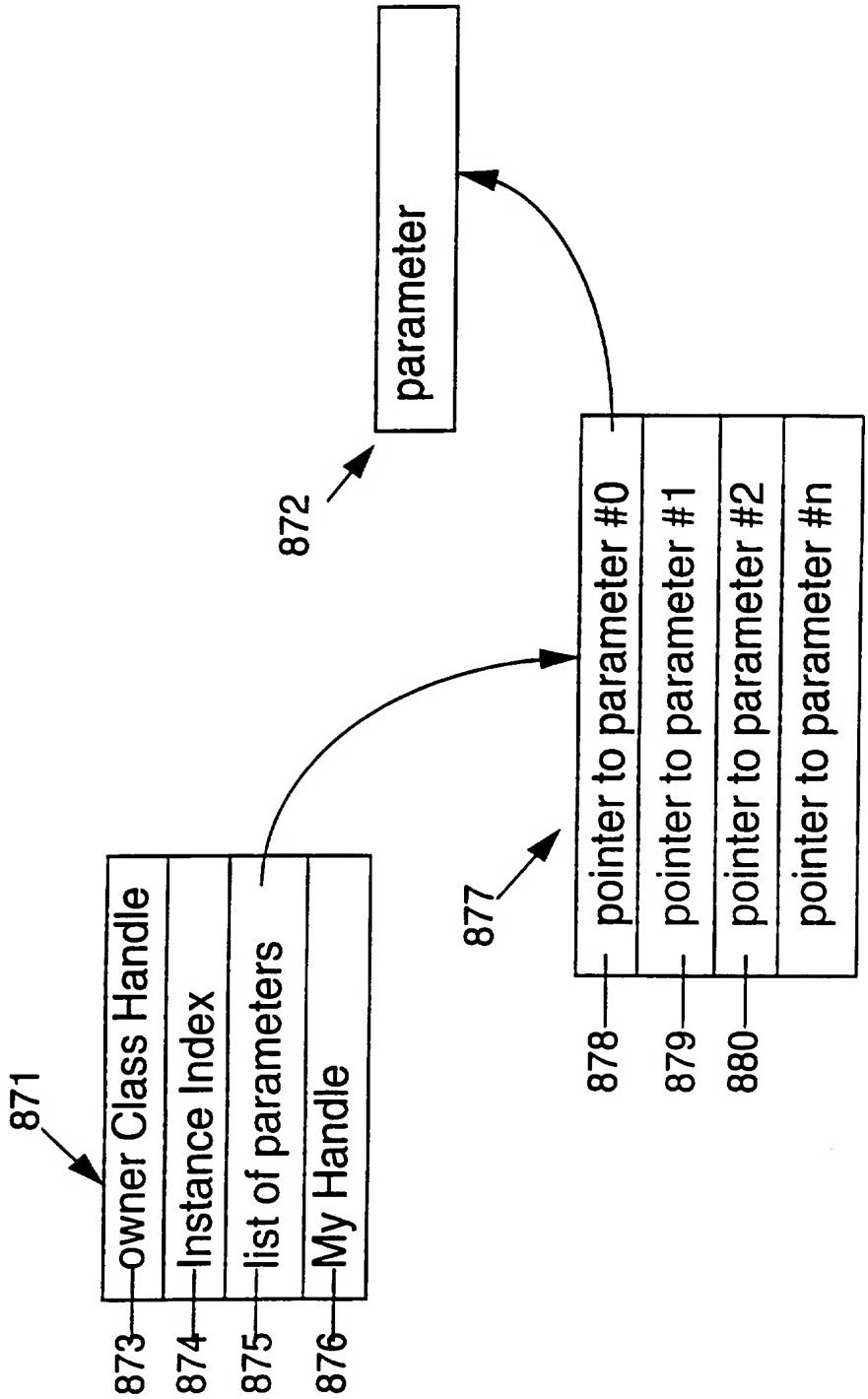

parameter
872
pointer to parameter #0
pointer to parameter #1
pointer to parameter #2
pointer to parameter #n
877
878
879
880
owner Class Handle
Instance Index
list of parameters
My Handle
871
873
874
875
876

FIG. 49

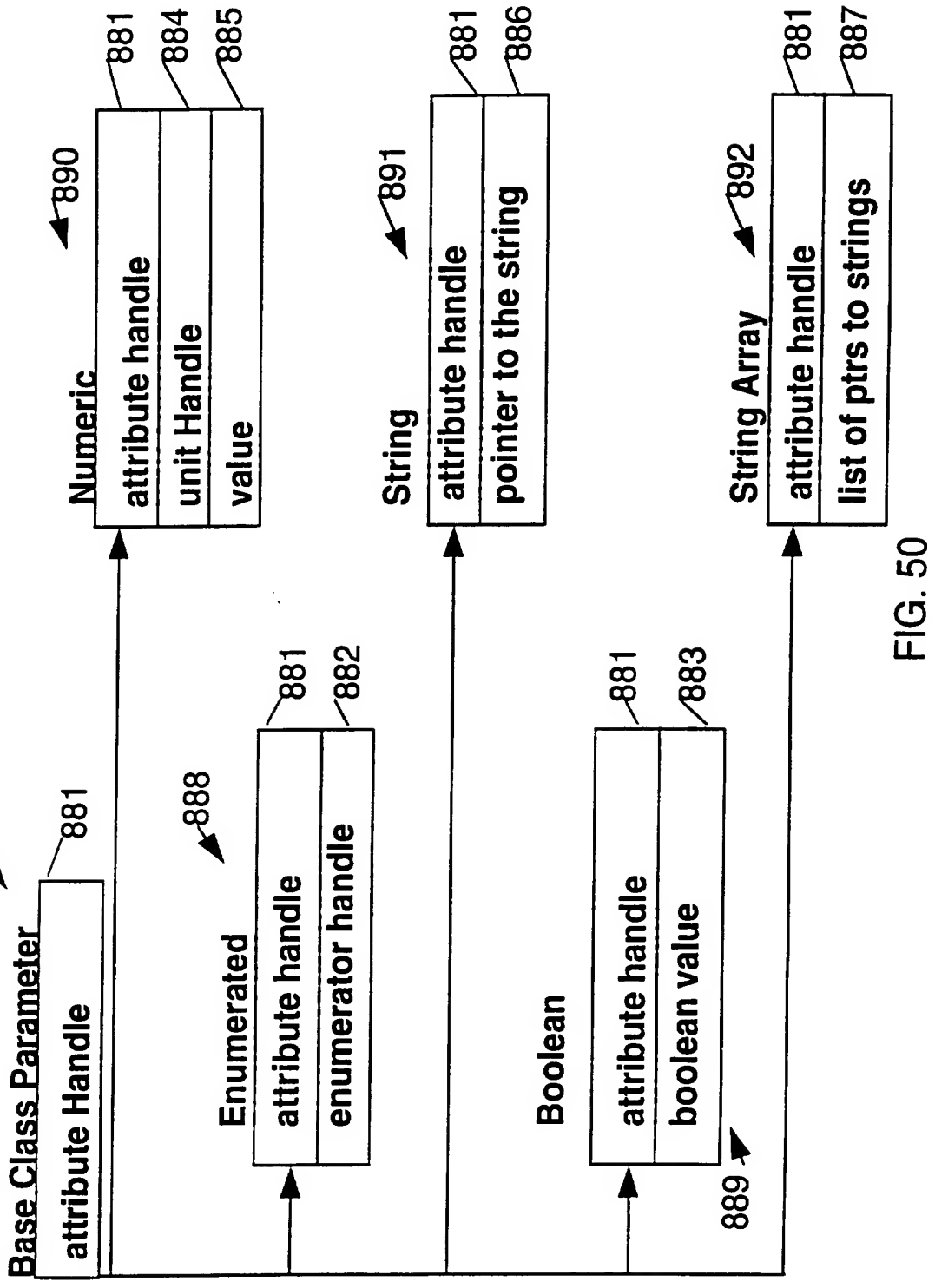

872
Base Class Parameter
attribute Handle
881
888
Enumerated
attribute handle
enumerator handle
881
882
Boolean
attribute handle
boolean value
881
883
889
890
Numeric
attribute handle
unit Handle
value
881
884
885
891
String
attribute handle
pointer to the string
881
886
892
String Array
attribute handle
list of ptrs to strings
881
887

FIG. 50

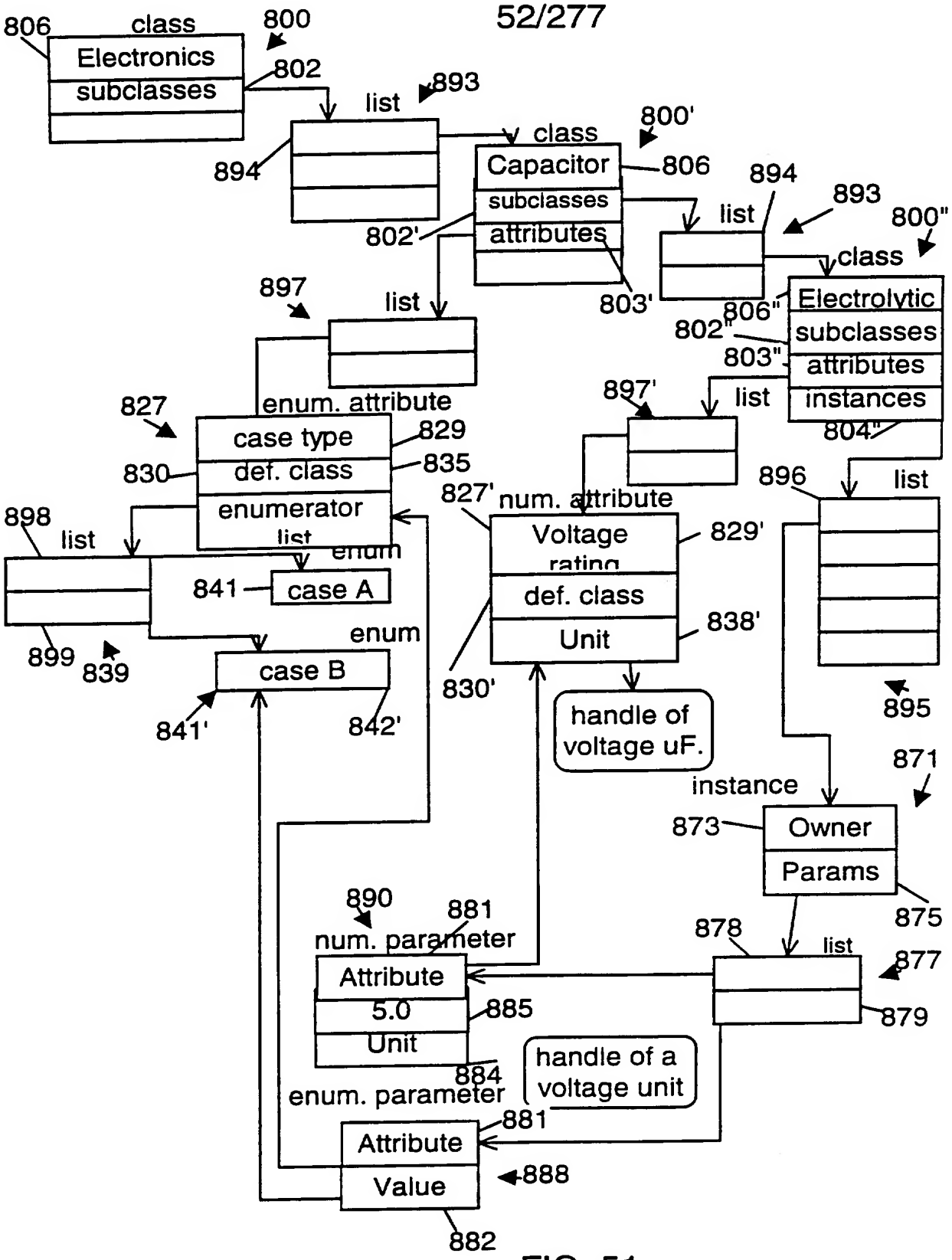

806
class
800
Electronics
802
subclasses
893
list
894
800'
class
Capacitor
806
subclasses
attributes
802'
803'
894
893
list
800"
class
Electrolytic
806"
802"
subclasses
803"
attributes
instances
804"
897
list
897'
list
827
enum. attribute
case type
829
830
def. class
835
enumerator
list
827'
num. attribute
Voltage
rating
829'
def. class
Unit
838'
830'
898
list
enum
841
case A
enum
case B
899
839
841'
842'
handle of
voltage uF.
896
list
895
871
instance
873
Owner
Params
875
890
num. parameter
881
Attribute
5.0
Unit
885
884
878
list
877
879
handle of a
voltage unit
enum. parameter
881
Attribute
Value
888
882

FIG. 51

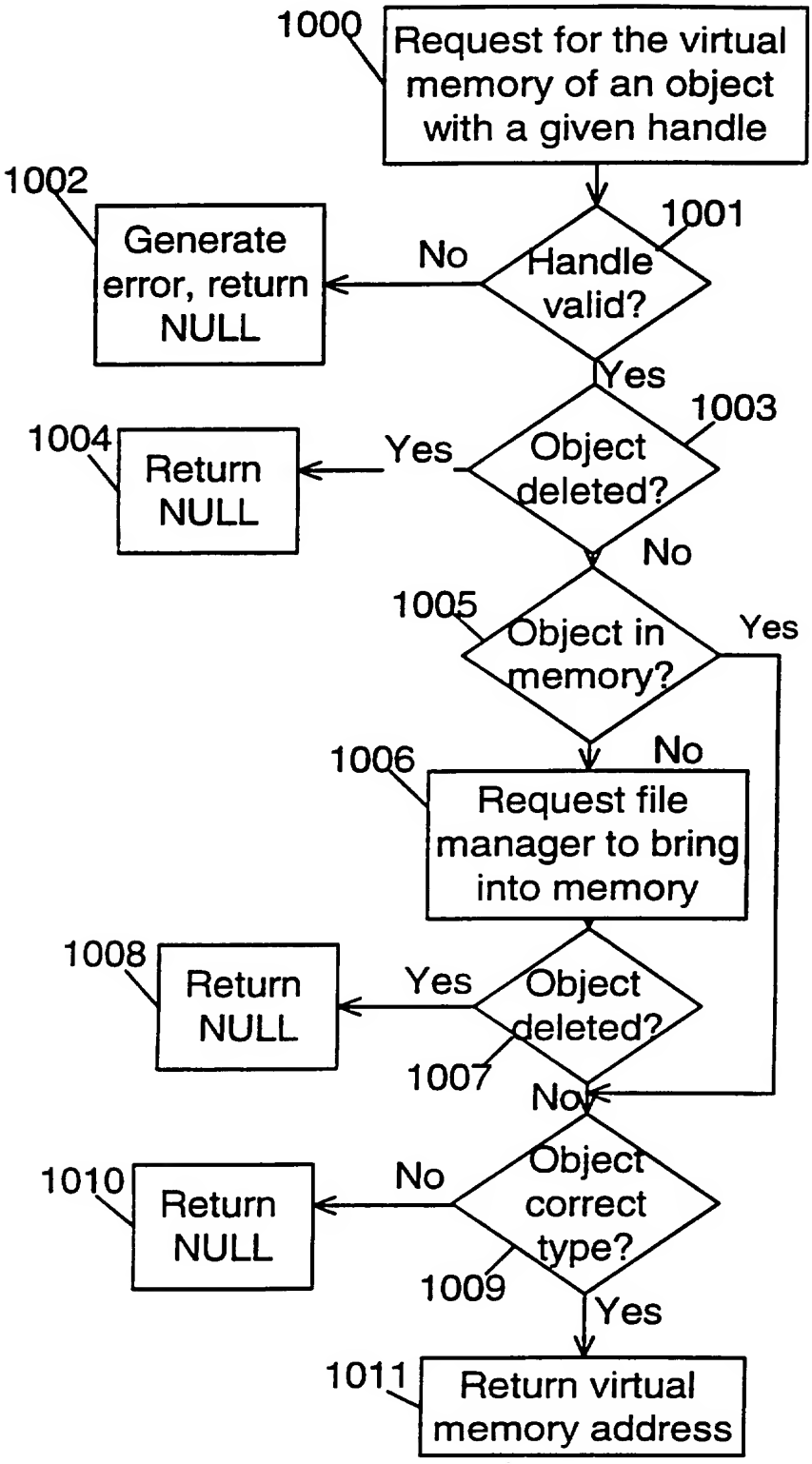

1000
Request for the virtual memory of an object with a given handle
1001
Handle valid?
No
1002
Generate error, return NULL
Yes
1003
Object deleted?
Yes
1004
Return NULL
No
1005
Object in memory?
Yes
No
1006
Request file manager to bring into memory
1008
Return NULL
Yes
Object deleted?
1007
No
Object correct type?
No
1010
Return NULL
1009
Yes
1011
Return virtual memory address

FIG. 52

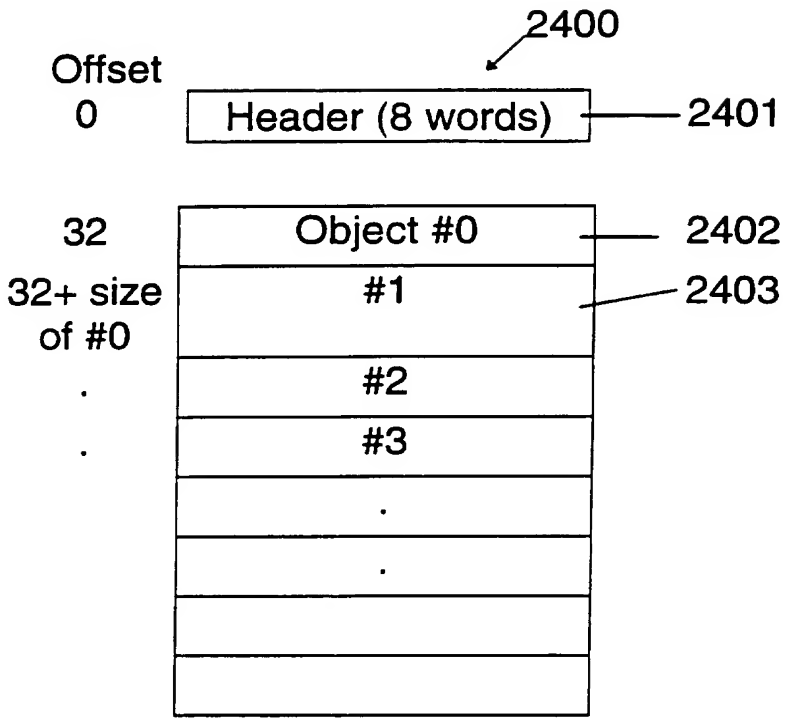

2400
Offset
0
Header (8 words)
2401
32
Object #0
2402
32+ size
of #0
#1
2403
.
#2
.
#3
.
.

FIG. 53

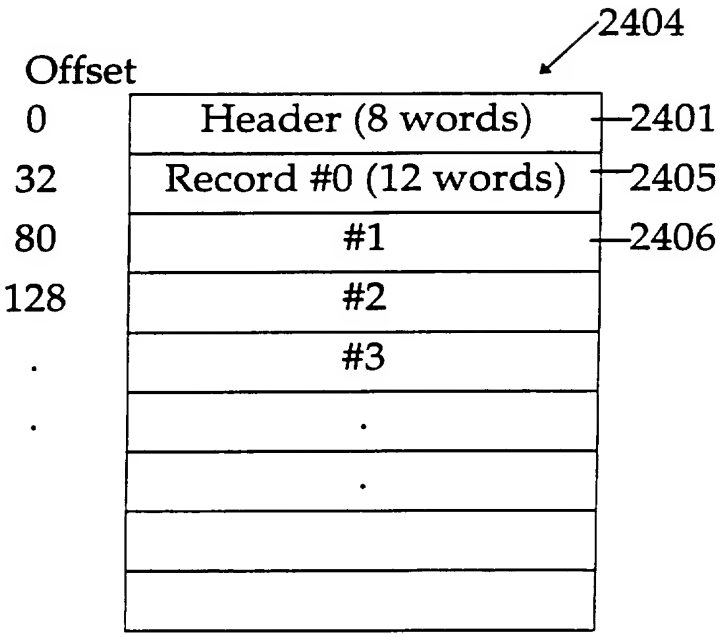

2404
Offset
0
Header (8 words)
2401
32
Record #0 (12 words)
2405
80
#1
2406
128
#2
.
#3
.
.
.

FIG. 54

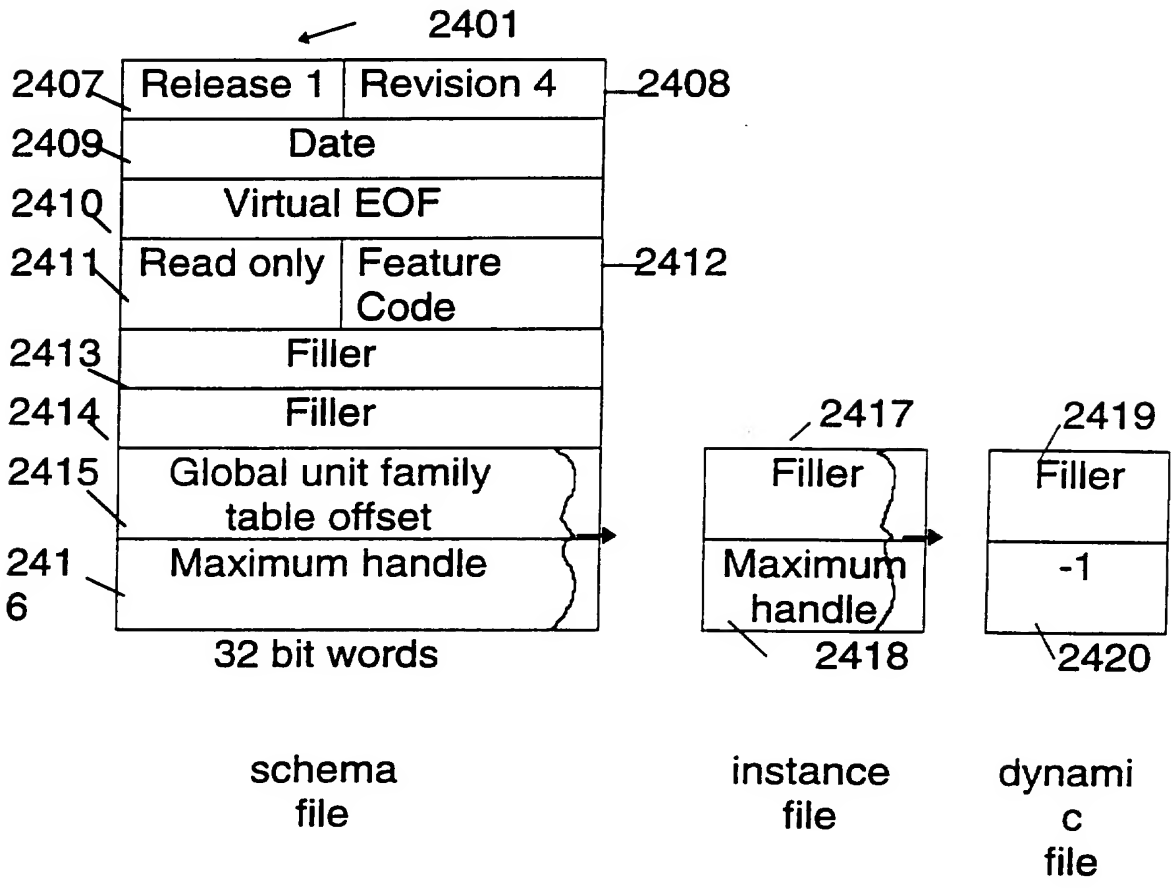

2401
2407
Release 1
Revision 4
2408
2409
Date
2410
Virtual EOF
2411
Read only
Feature Code
2412
2413
Filler
2414
Filler
2415
Global unit family table offset
241 6
Maximum handle
32 bit words
2417
Filler
Maximum handle
2418
2419
Filler
-1
2420
schema file
instance file
dynami c file

FIG. 55

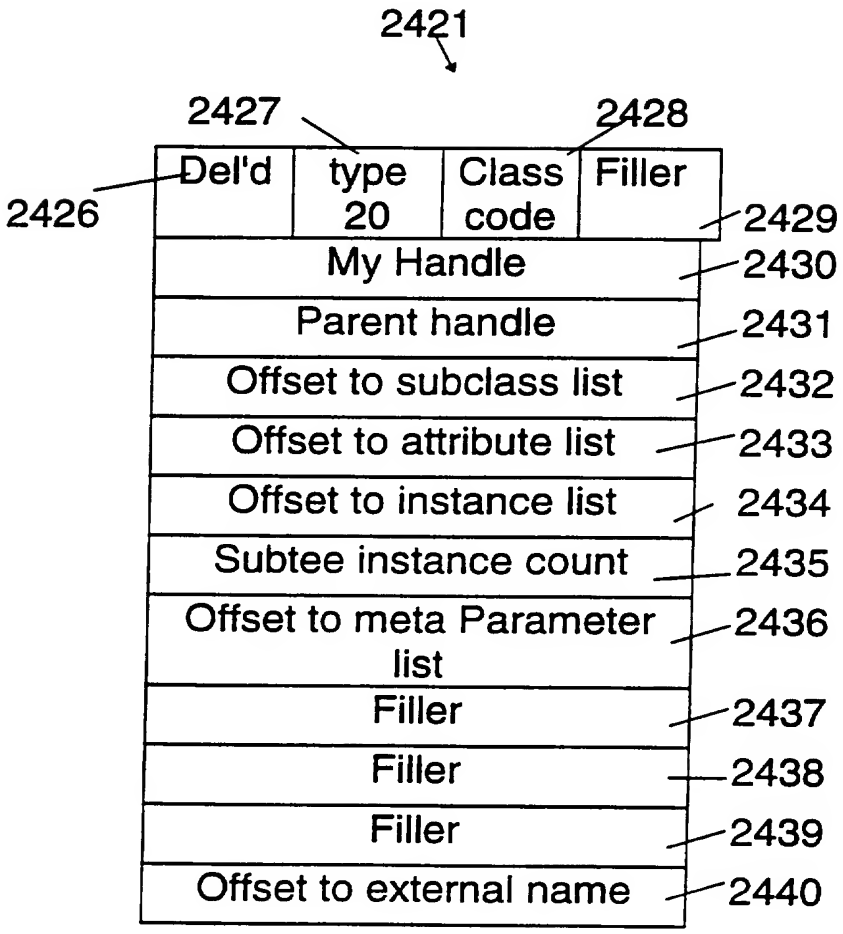

2421
2427
2428
Del'd
type 20
Class code
Filler
2426
2429
My Handle
2430
Parent handle
2431
Offset to subclass list
2432
Offset to attribute list
2433
Offset to instance list
2434
Subtee instance count
2435
Offset to meta Parameter list
2436
Filler
2437
Filler
2438
Filler
2439
Offset to external name
2440

FIG. 56

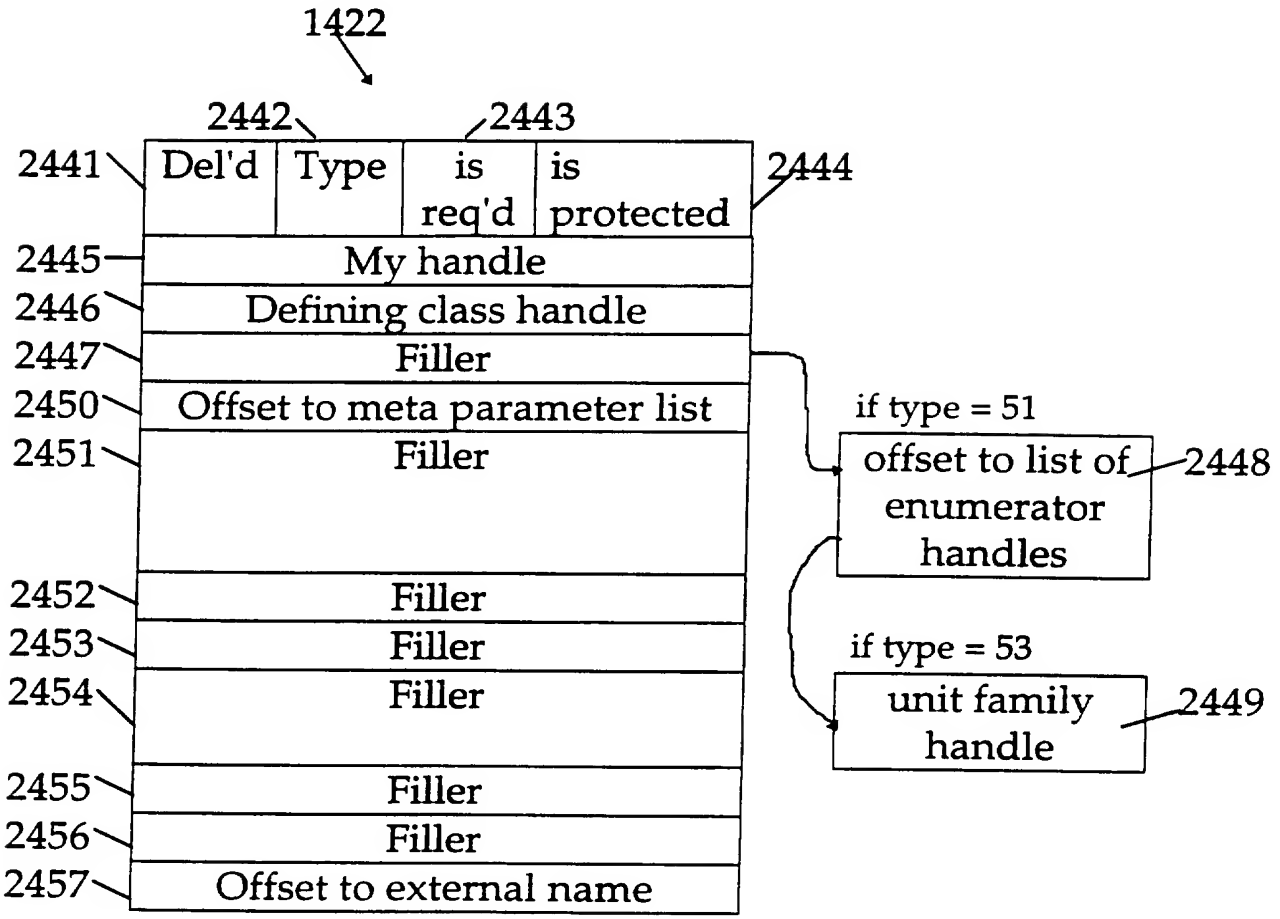

1422
2442
2443
2441
Del'd
Type
is req'd
is protected
2444
2445
My handle
2446
Defining class handle
2447
Filler
2450
Offset to meta parameter list
2451
Filler
2452
Filler
2453
Filler
2454
Filler
2455
Filler
2456
Filler
2457
Offset to external name
if type = 51
offset to list of enumerator handles
2448
if type = 53
unit family handle
2449

FIG. 57

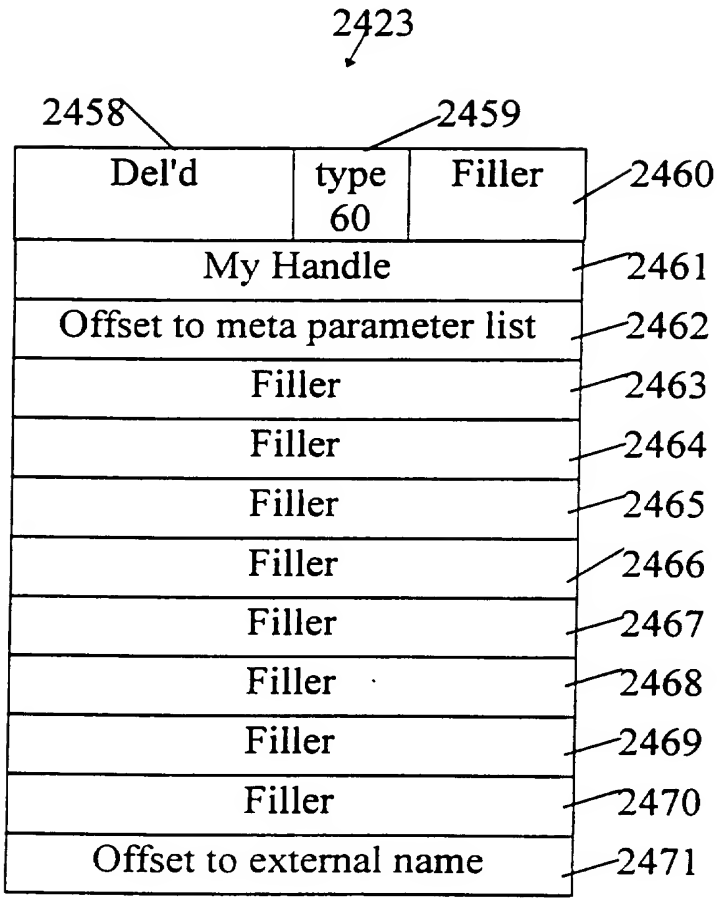

2423
2458
2459
Del'd
type 60
Filler
2460
My Handle
2461
Offset to meta parameter list
2462
Filler
2463
Filler
2464
Filler
2465
Filler
2466
Filler
2467
Filler
2468
Filler
2469
Filler
2470
Offset to external name
2471

FIG. 58

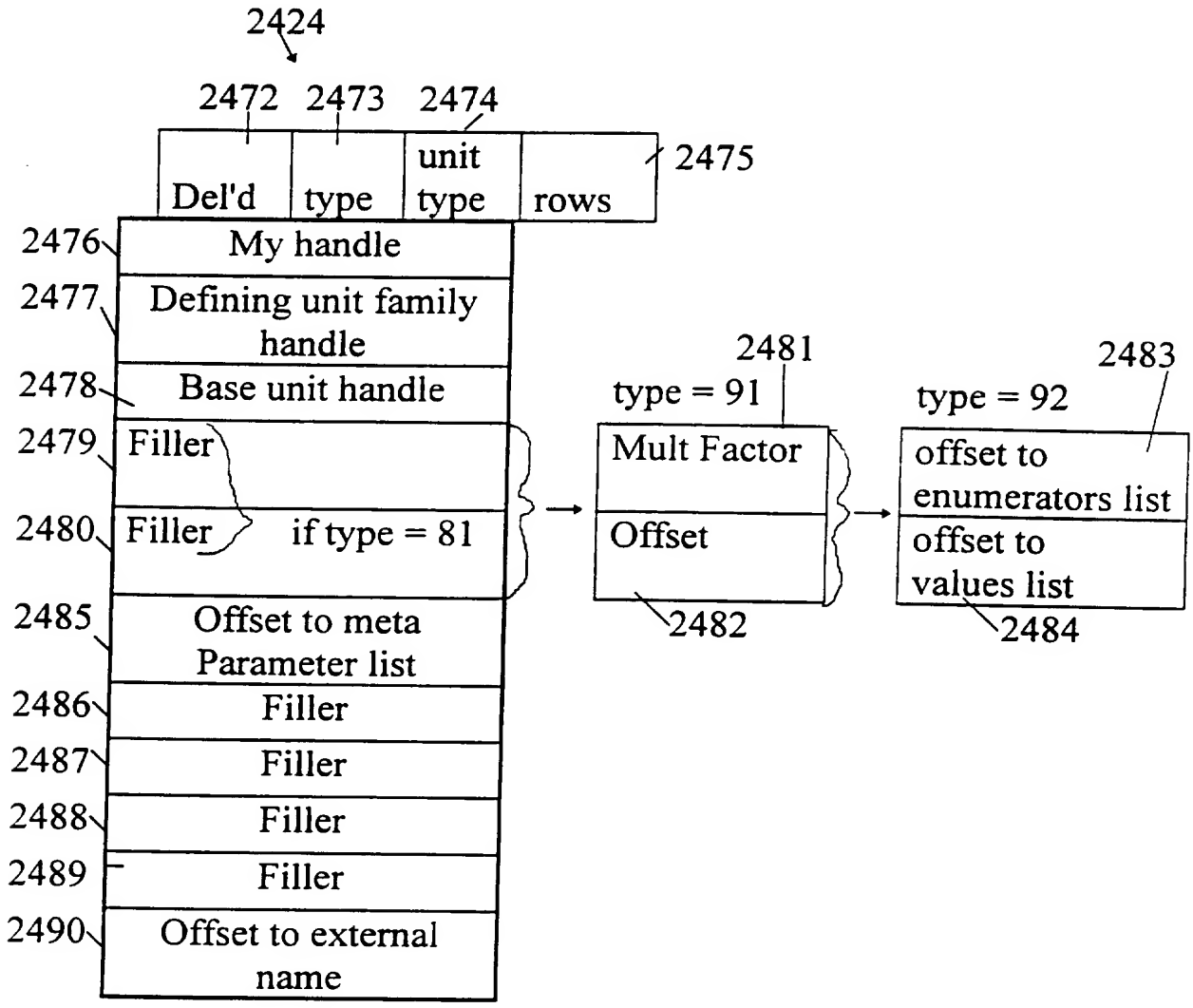

2424
2472
2473
2474
Del'd
type
unit type
rows
2475
2476
My handle
2477
Defining unit family handle
2478
Base unit handle
2479
Filler
2480
Filler
if type = 81
2485
Offset to meta Parameter list
2486
Filler
2487
Filler
2488
Filler
2489
Filler
2490
Offset to external name
2481
type = 91
Mult Factor
Offset
2482
2483
type = 92
offset to enumerators list
offset to values list
2484

FIG. 59

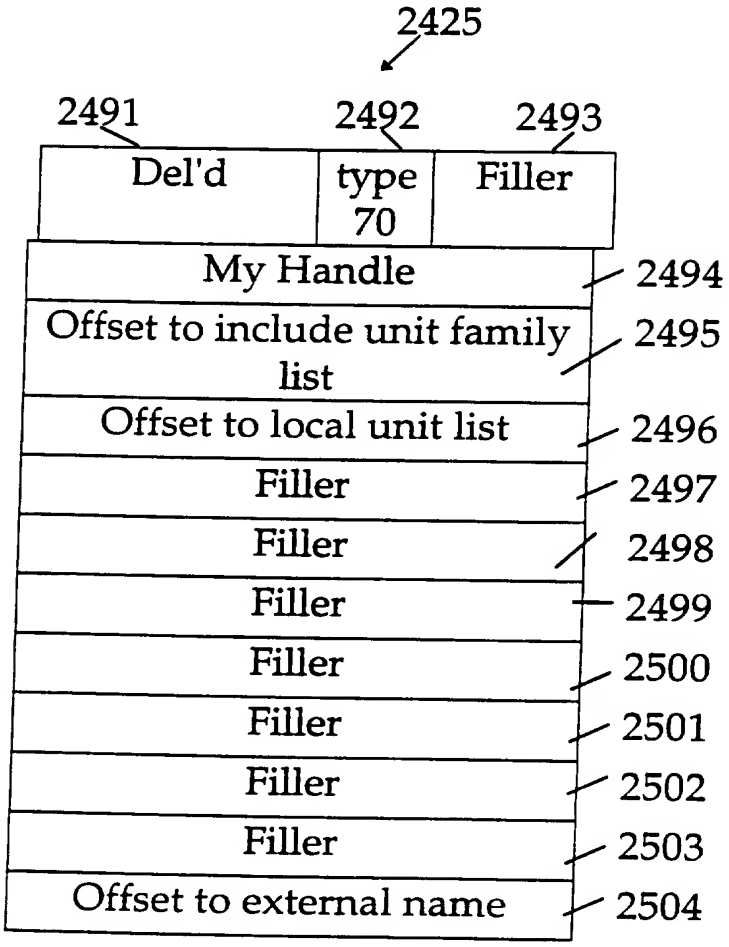

2425
2491
2492
2493
Del'd
type 70
Filler
My Handle
2494
Offset to include unit family list
2495
Offset to local unit list
2496
Filler
2497
Filler
2498
Filler
2499
Filler
2500
Filler
2501
Filler
2502
Filler
2503
Offset to external name
2504

FIG. 60

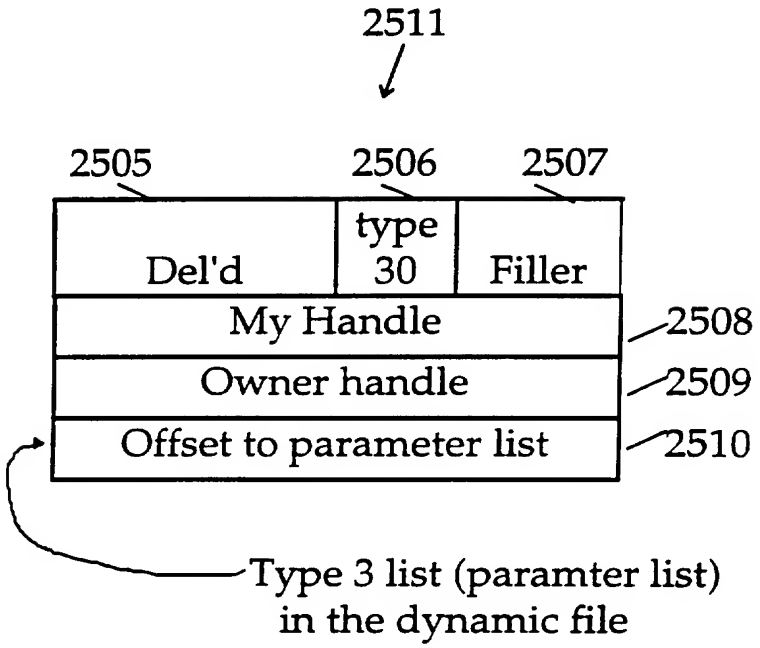
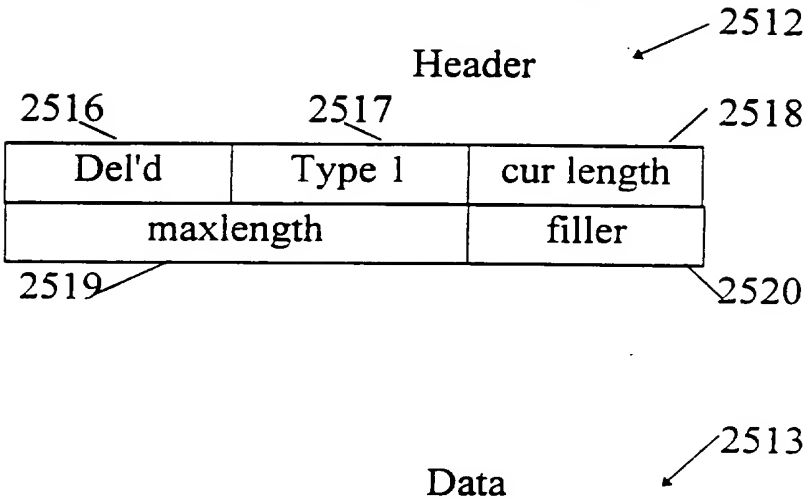

2511
2505
2506
2507
Del'd
type 30
Filler
My Handle
2508
Owner handle
2509
Offset to parameter list
2510
Type 3 list (paramter list) in the dynamic file

FIG. 61

Type 1 - string



C	a	p	a
c	i	t	o
r	s	\0	\0

FIG. 62

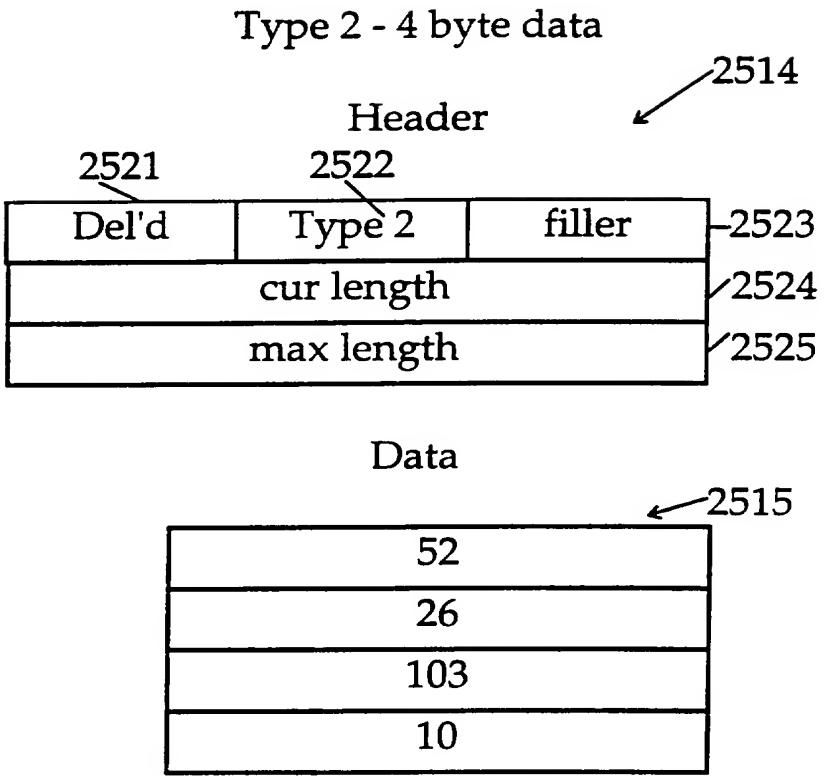

Type 2 - 4 byte data
2514
Header
2521
2522
Del'd
Type 2
filler
2523
cur length
2524
max length
2525
Data
2515
52
26
103
10

FIG. 63

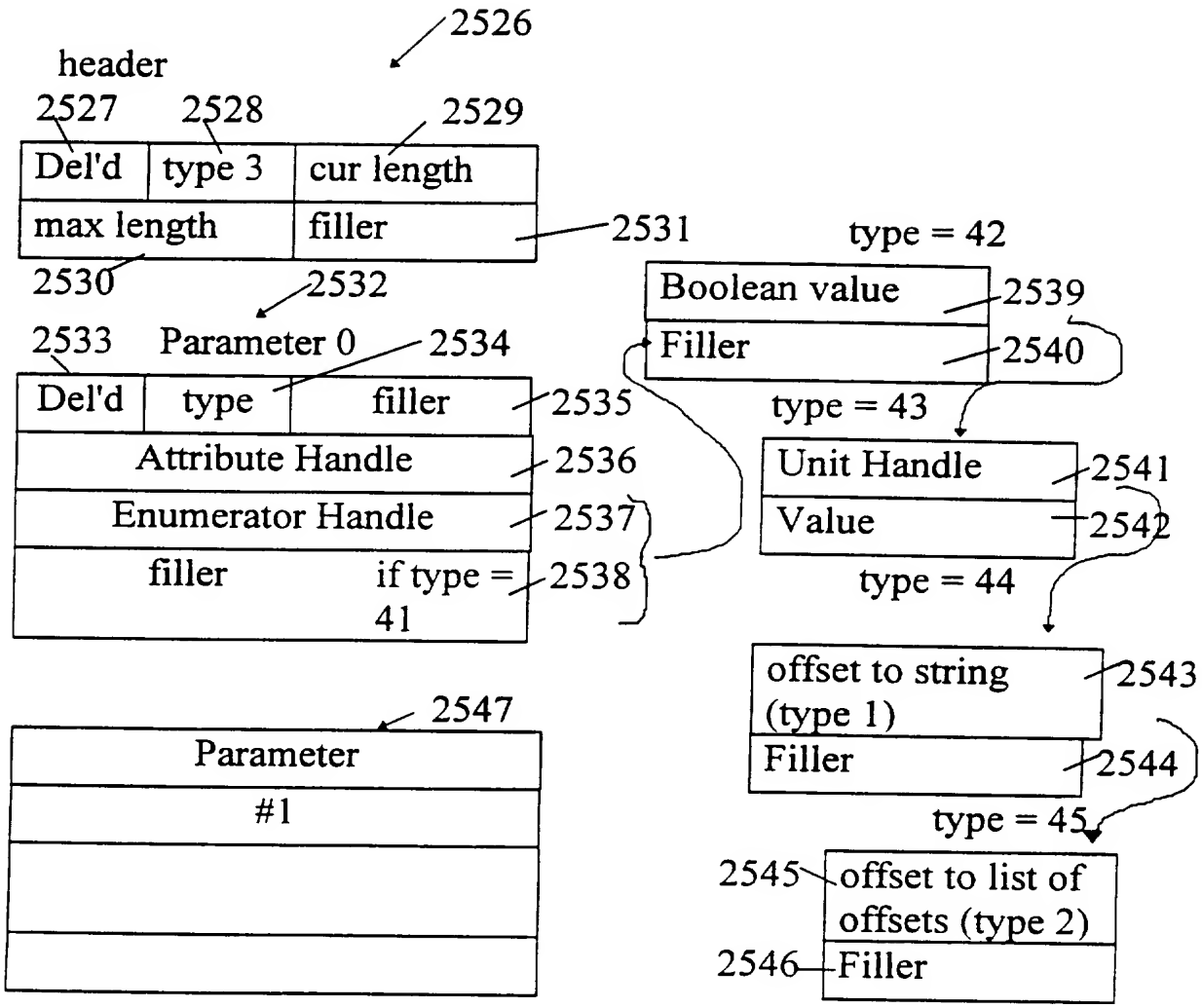

2526
header
2527
2528
2529
Del'd
type 3
cur length
max length
filler
2531
2530
2532
2533
Parameter 0
2534
Del'd
type
filler
2535
Attribute Handle
2536
Enumerator Handle
2537
filler
if type = 41
2538
2547
Parameter
#1
type = 42
Boolean value
2539
Filler
2540
type = 43
Unit Handle
2541
Value
2542
type = 44
offset to string (type 1)
2543
Filler
2544
type = 45
2545
offset to list of offsets (type 2)
2546
Filler

FIG. 64

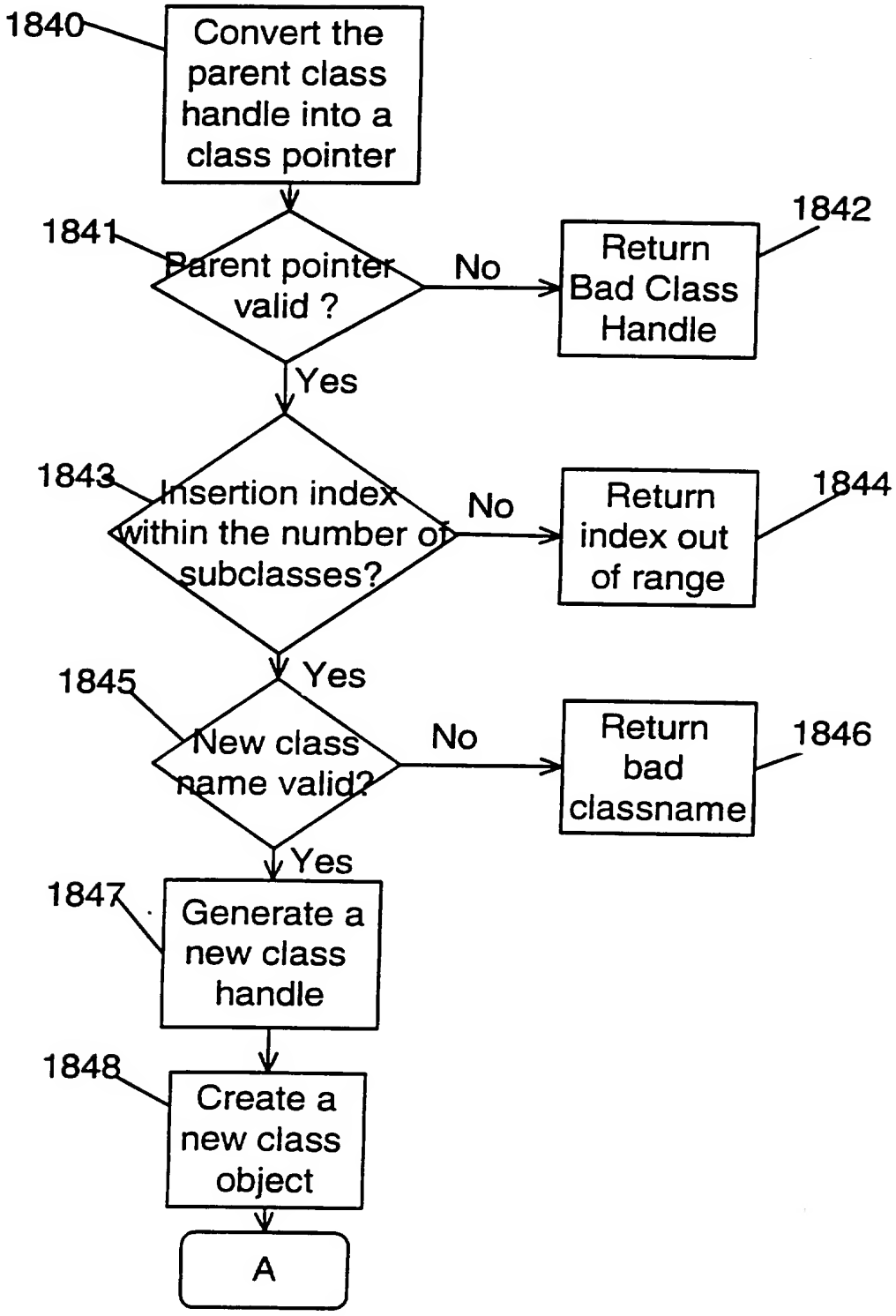

1840
Convert the parent class handle into a class pointer
1841
Parent pointer valid ?
No
1842
Return Bad Class Handle
Yes
1843
Insertion index within the number of subclasses?
No
1844
Return index out of range
Yes
1845
New class name valid?
No
1846
Return bad classname
Yes
1847
Generate a new class handle
1848
Create a new class object
A

FIG. 65

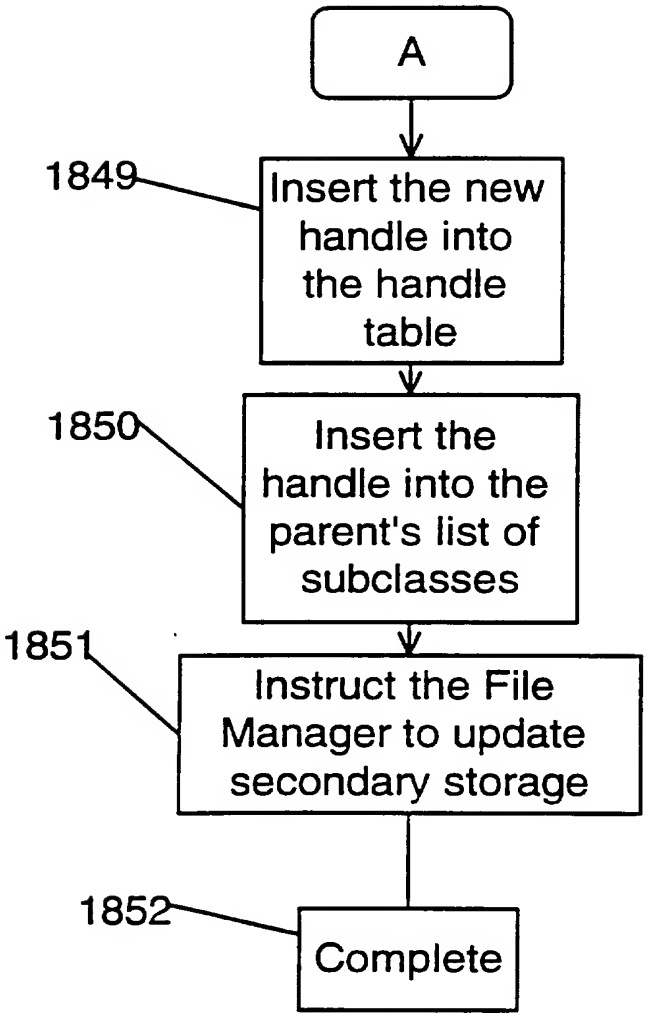

A
1849
Insert the new handle into the handle table
1850
Insert the handle into the parent's list of subclasses
1851
Instruct the File Manager to update secondary storage
1852
Complete

FIG. 66

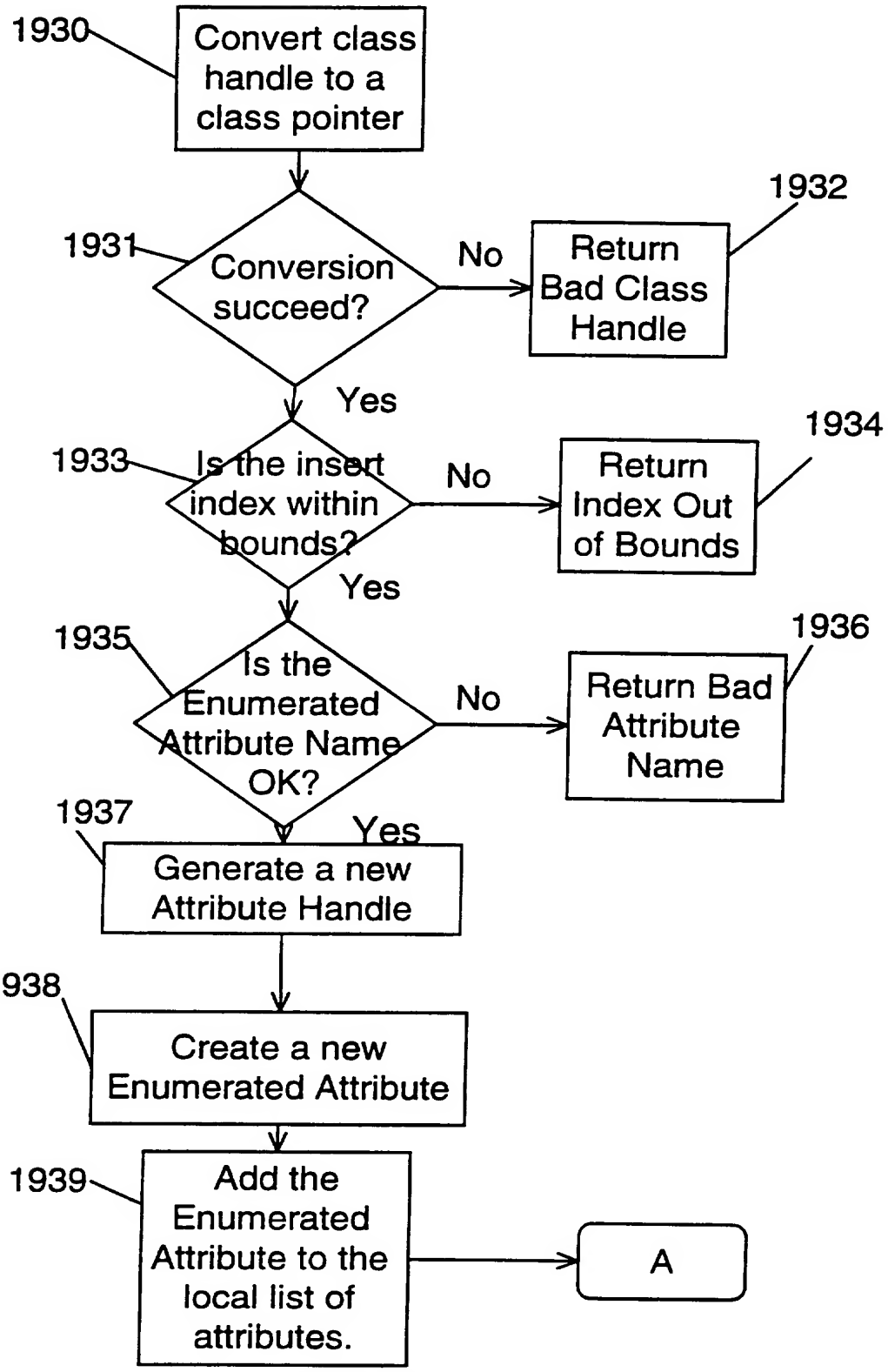

1930
Convert class handle to a class pointer
1931
Conversion succeed?
No
1932
Return Bad Class Handle
Yes
1933
Is the insert index within bounds?
No
1934
Return Index Out of Bounds
Yes
1935
Is the Enumerated Attribute Name OK?
No
1936
Return Bad Attribute Name
Yes
1937
Generate a new Attribute Handle
1938
Create a new Enumerated Attribute
1939
Add the Enumerated Attribute to the local list of attributes.
A

FIG. 67

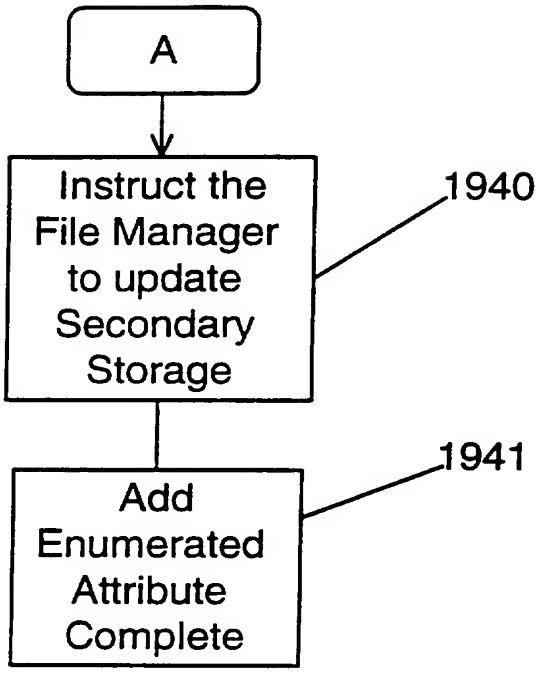

A
Instruct the File Manager to update Secondary Storage
1940
Add Enumerated Attribute Complete
1941

FIG. 68

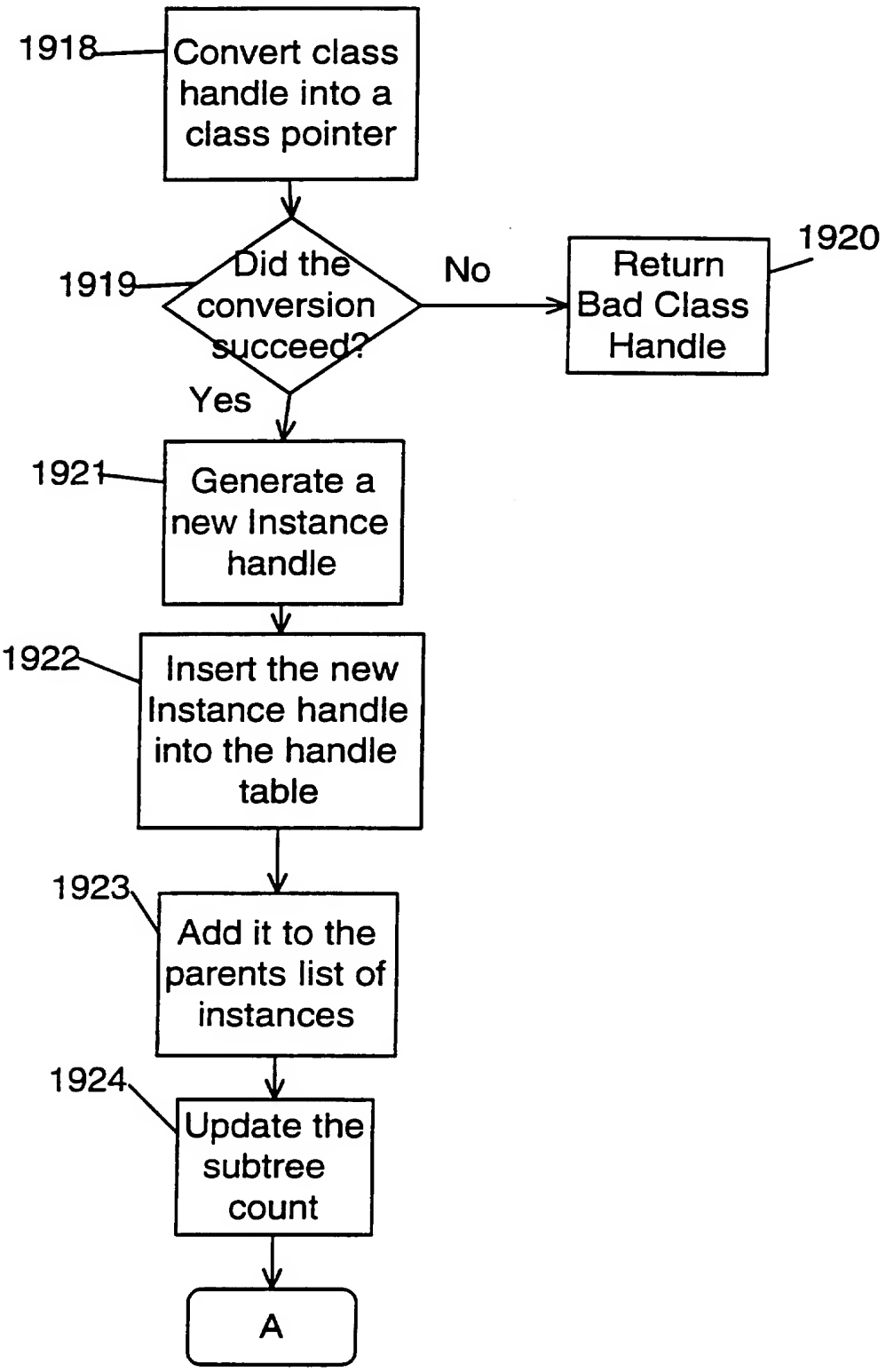

1918
Convert class handle into a class pointer
1919
Did the conversion succeed?
No
1920
Return Bad Class Handle
Yes
1921
Generate a new Instance handle
1922
Insert the new Instance handle into the handle table
1923
Add it to the parents list of instances
1924
Update the subtree count
A

FIG. 69

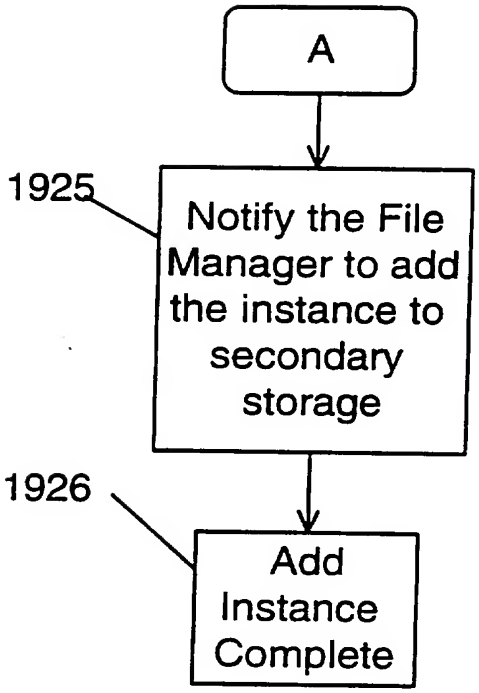

A
1925
Notify the File Manager to add the instance to secondary storage
1926
Add Instance Complete

FIG. 70

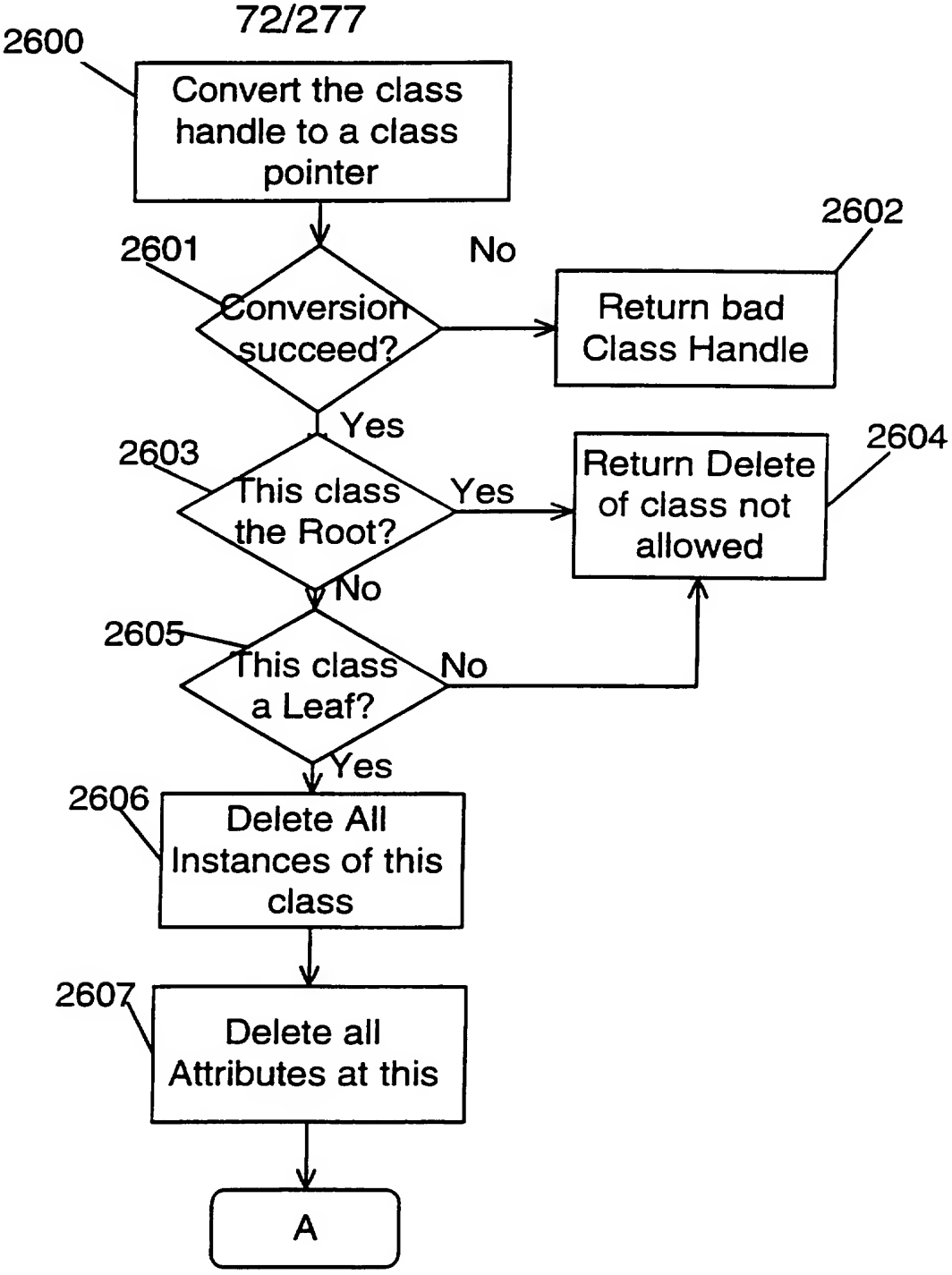

2600
Convert the class handle to a class pointer
2601
Conversion succeed?
No
2602
Return bad Class Handle
Yes
2603
This class the Root?
Yes
2604
Return Delete of class not allowed
No
2605
This class a Leaf?
No
Yes
2606
Delete All Instances of this class
2607
Delete all Attributes at this
A

FIG. 71

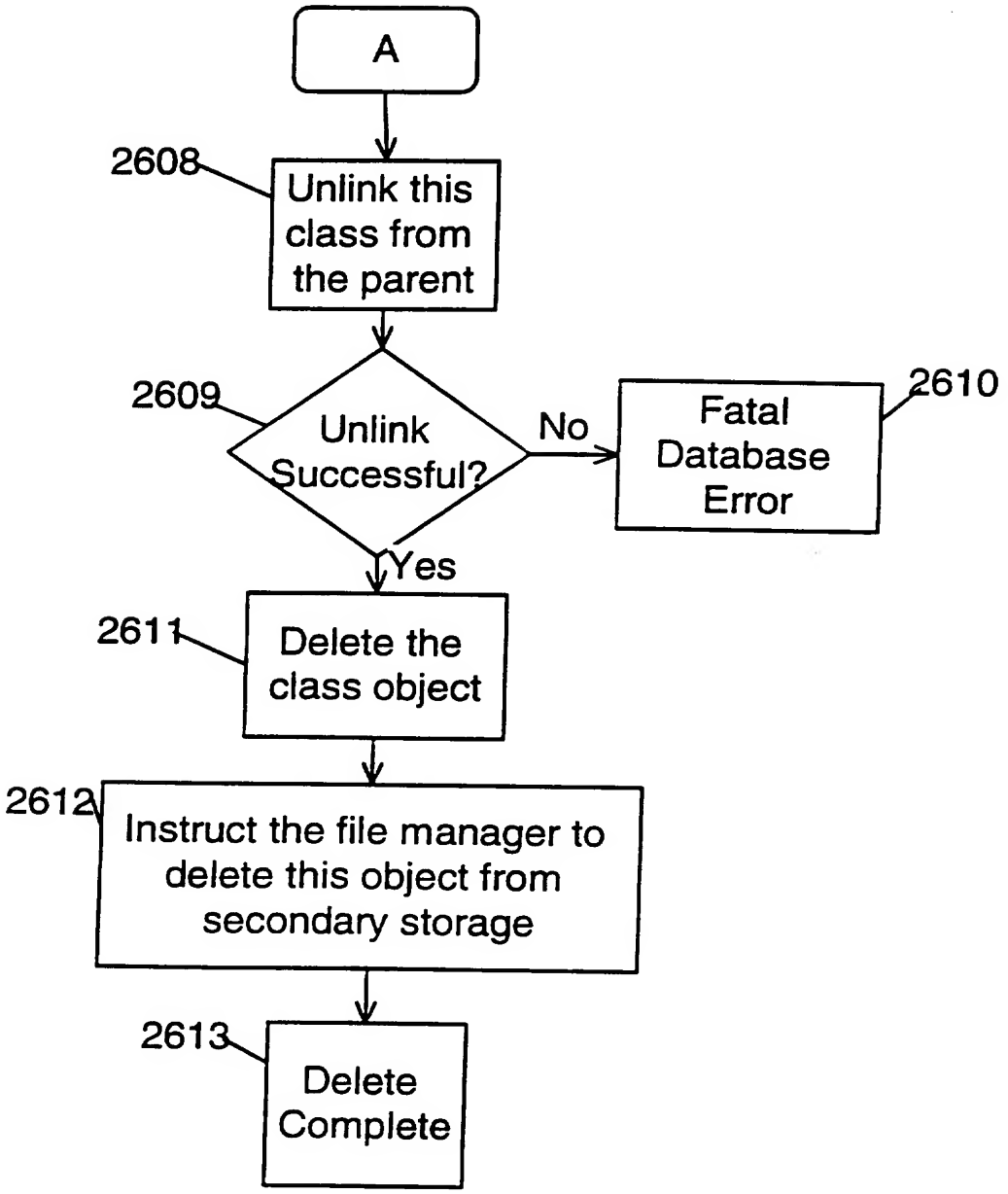

A
2608
Unlink this class from the parent
2609
Unlink Successful?
No
2610
Fatal Database Error
Yes
2611
Delete the class object
2612
Instruct the file manager to delete this object from secondary storage
2613
Delete Complete

FIG. 72

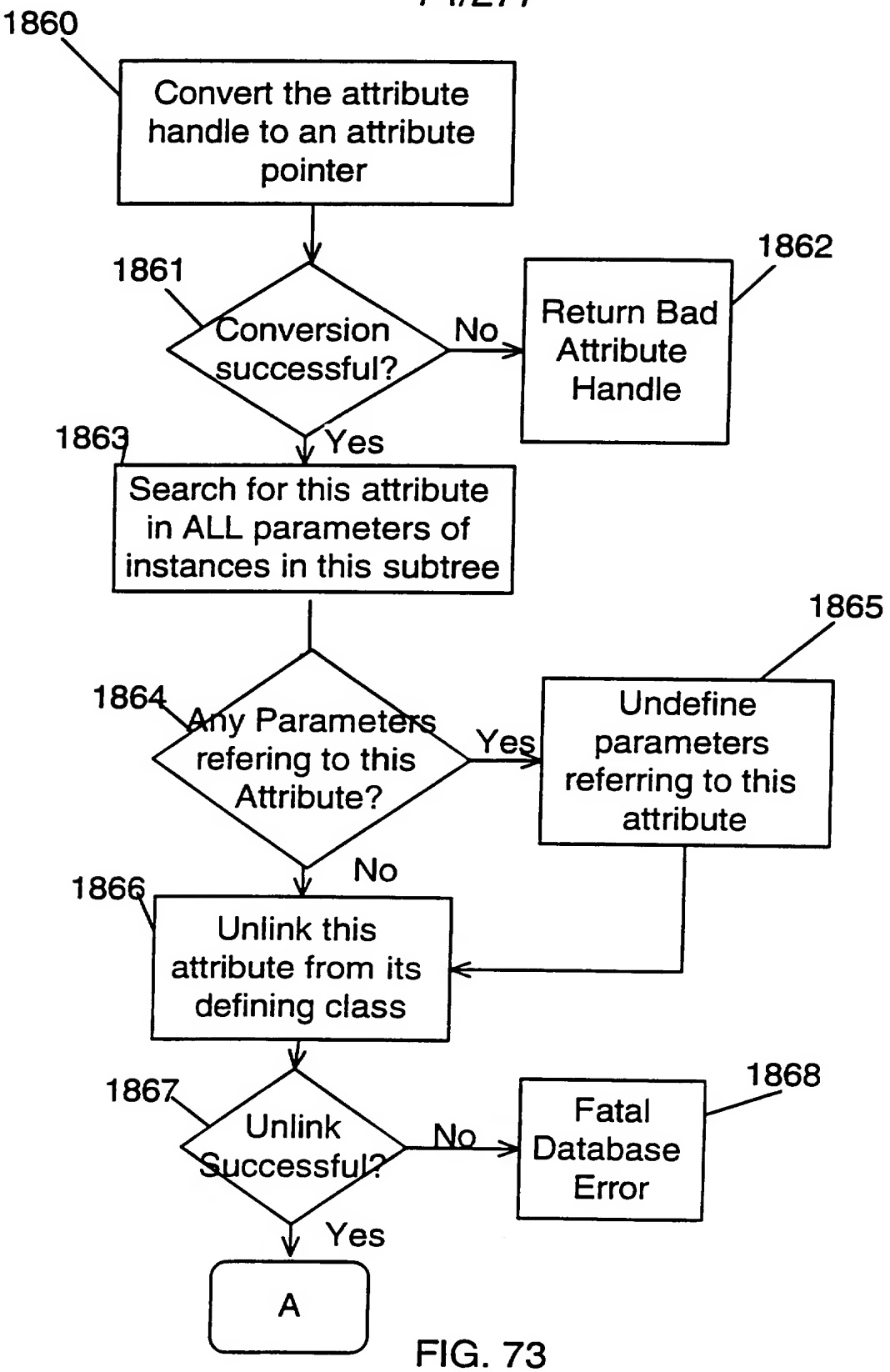

1860
Convert the attribute handle to an attribute pointer
1861
Conversion successful?
No
1862
Return Bad Attribute Handle
Yes
1863
Search for this attribute in ALL parameters of instances in this subtree
1865
1864
Any Parameters refering to this Attribute?
Yes
Undefine parameters referring to this attribute
No
1866
Unlink this attribute from its defining class
1867
Unlink Successful?
No
1868
Fatal Database Error
Yes
A

FIG. 73

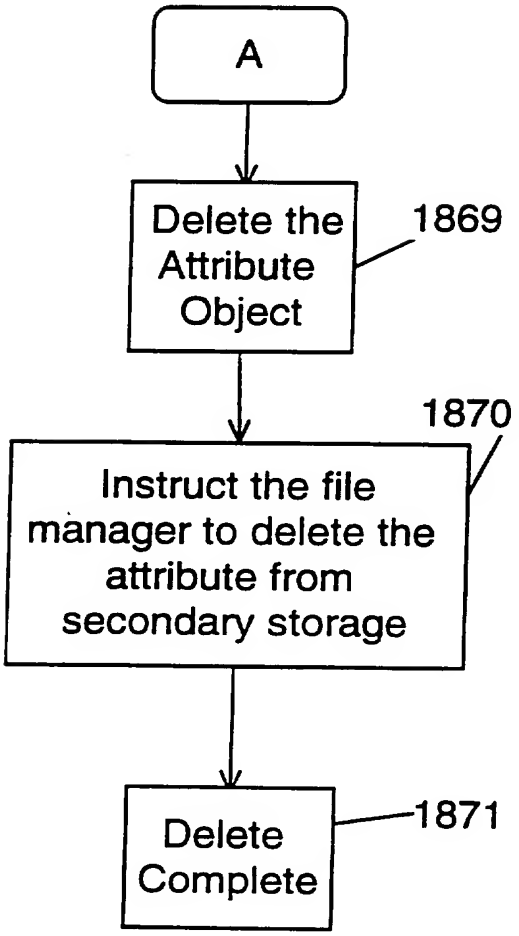

A
Delete the Attribute Object
1869
1870
Instruct the file manager to delete the attribute from secondary storage
Delete Complete
1871

FIG. 74

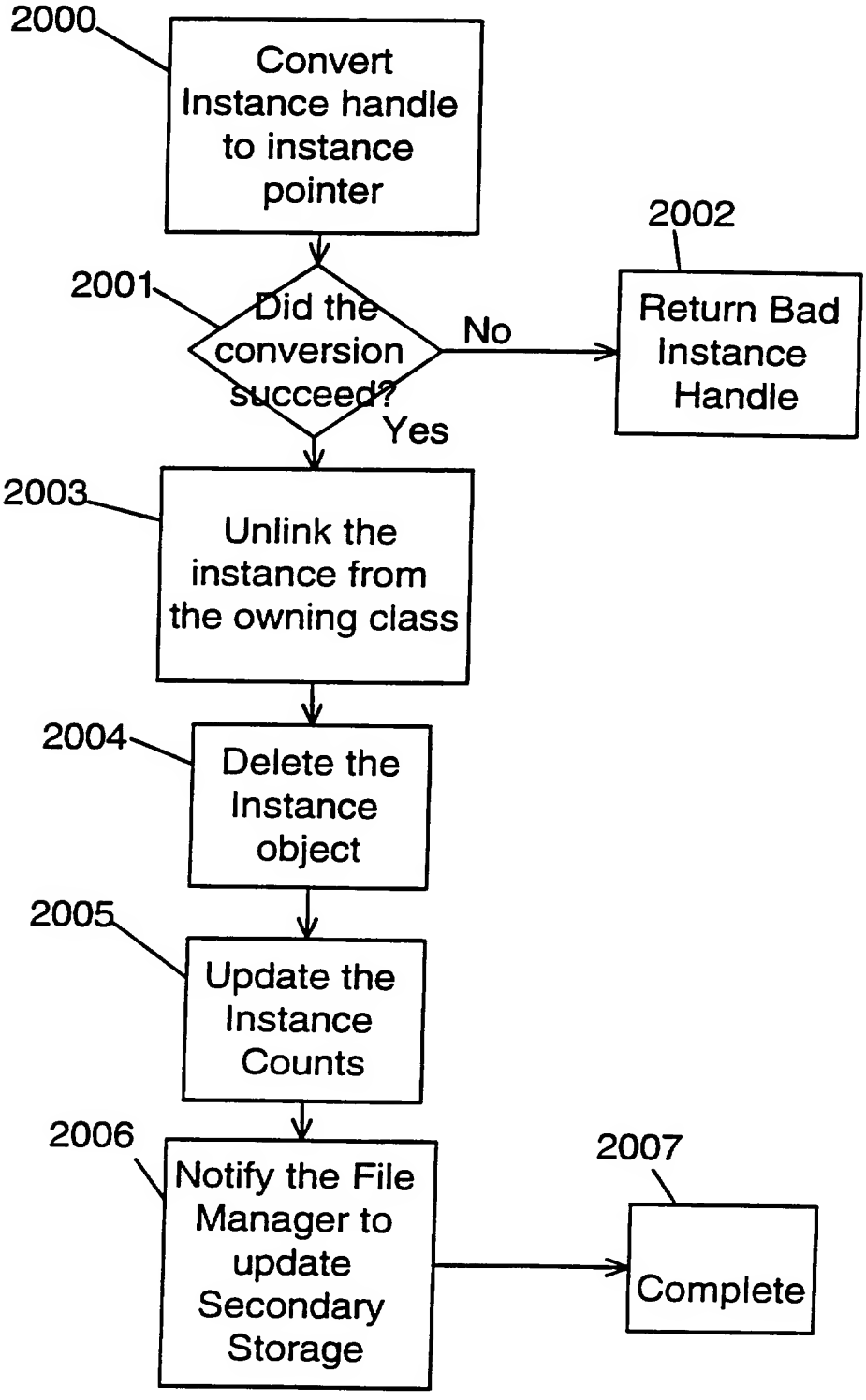

2000
Convert Instance handle to instance pointer
2001
Did the conversion succeed?
No
2002
Return Bad Instance Handle
Yes
2003
Unlink the instance from the owning class
2004
Delete the Instance object
2005
Update the Instance Counts
2006
Notify the File Manager to update Secondary Storage
2007
Complete

FIG. 75

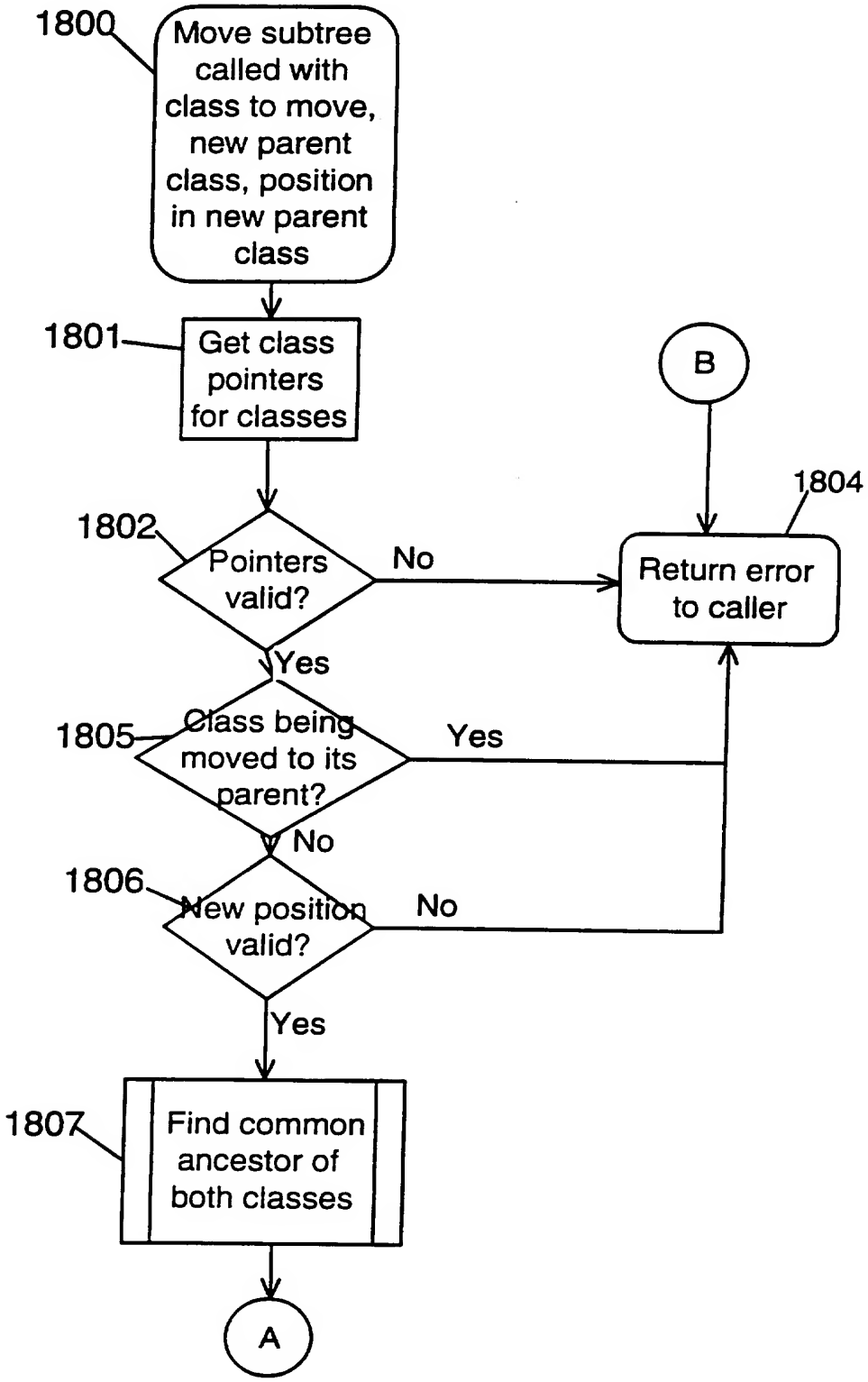

1800
Move subtree called with class to move, new parent class, position in new parent class
1801
Get class pointers for classes
B
1804
1802
Pointers valid?
No
Return error to caller
Yes
1805
Class being moved to its parent?
Yes
No
1806
New position valid?
No
Yes
1807
Find common ancestor of both classes
A

FIG. 76

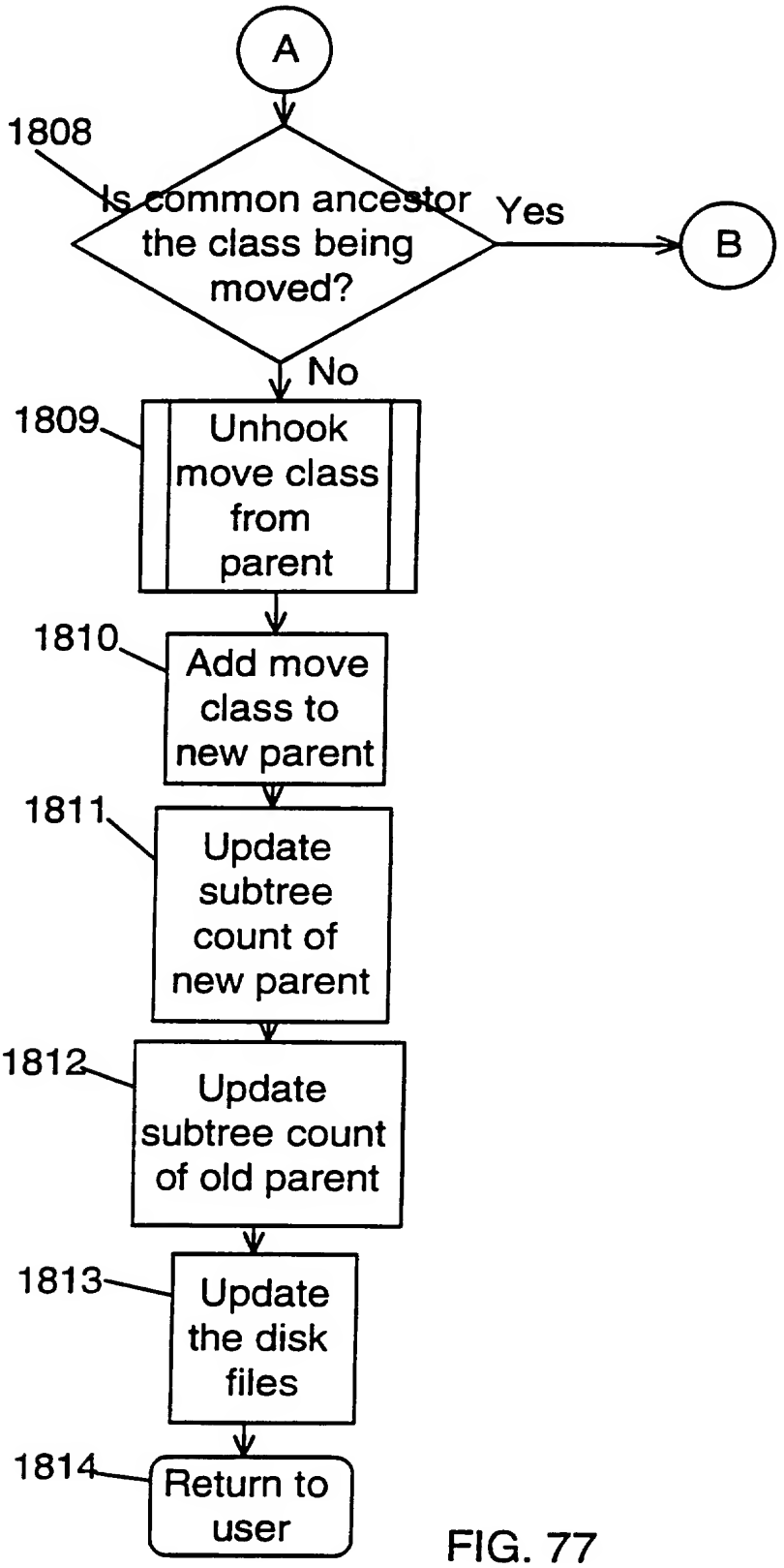

A
1808
Is common ancestor the class being moved?
Yes
B
No
1809
Unhook move class from parent
1810
Add move class to new parent
1811
Update subtree count of new parent
1812
Update subtree count of old parent
1813
Update the disk files
1814
Return to user

FIG. 77

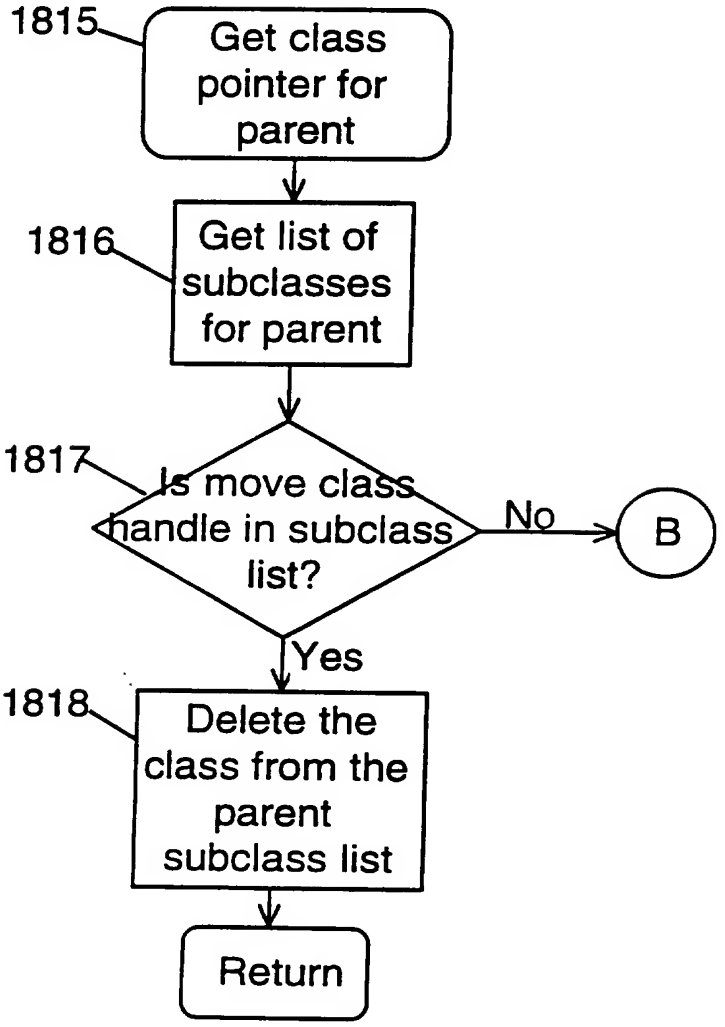

1815
Get class pointer for parent
1816
Get list of subclasses for parent
1817
Is move class handle in subclass list?
No
B
Yes
1818
Delete the class from the parent subclass list
Return

FIG. 78

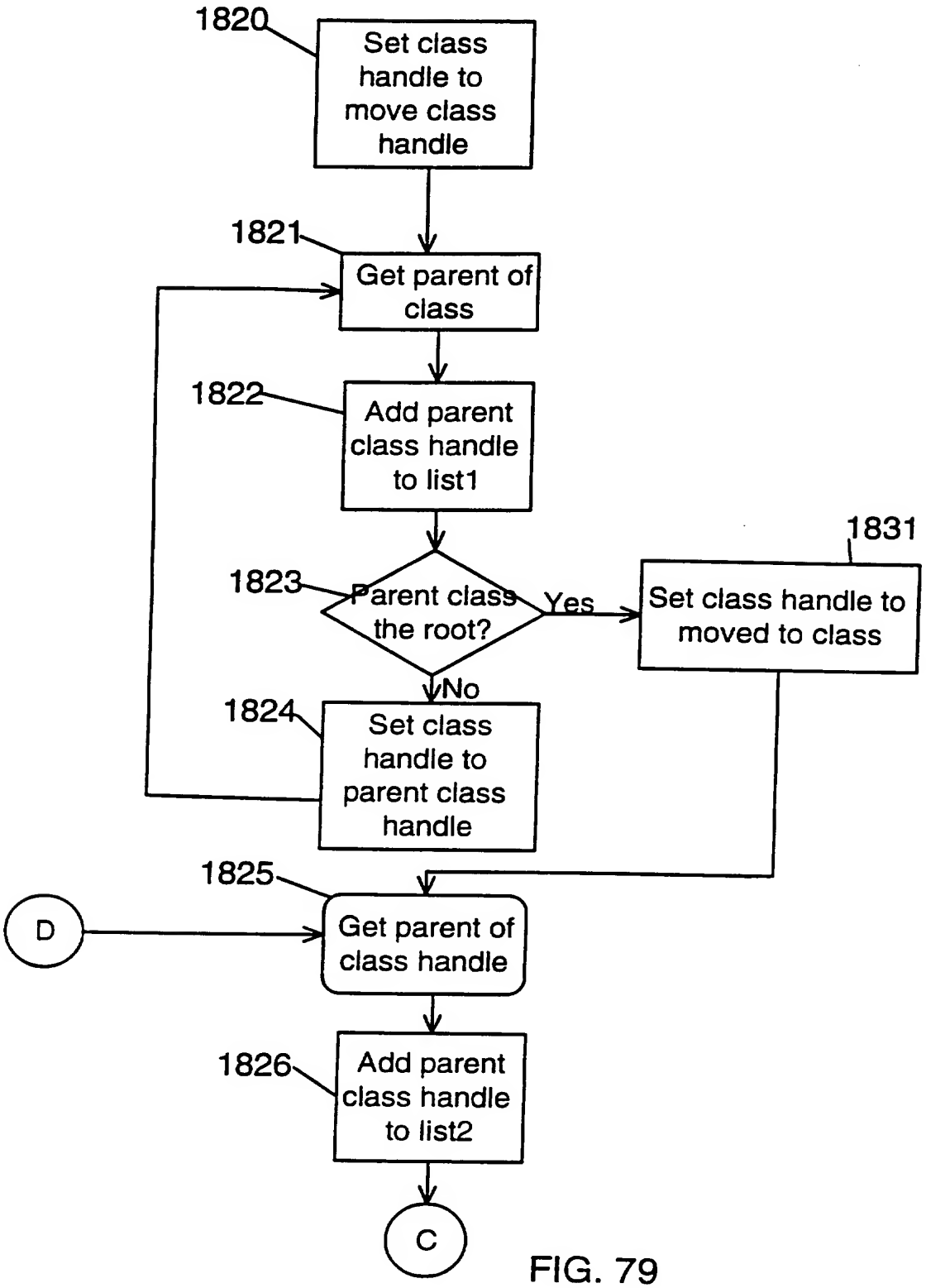

1820
Set class handle to move class handle
1821
Get parent of class
1822
Add parent class handle to list1
1823
Parent class the root?
Yes
1831
Set class handle to moved to class
No
1824
Set class handle to parent class handle
1825
D
Get parent of class handle
1826
Add parent class handle to list2
C

FIG. 79

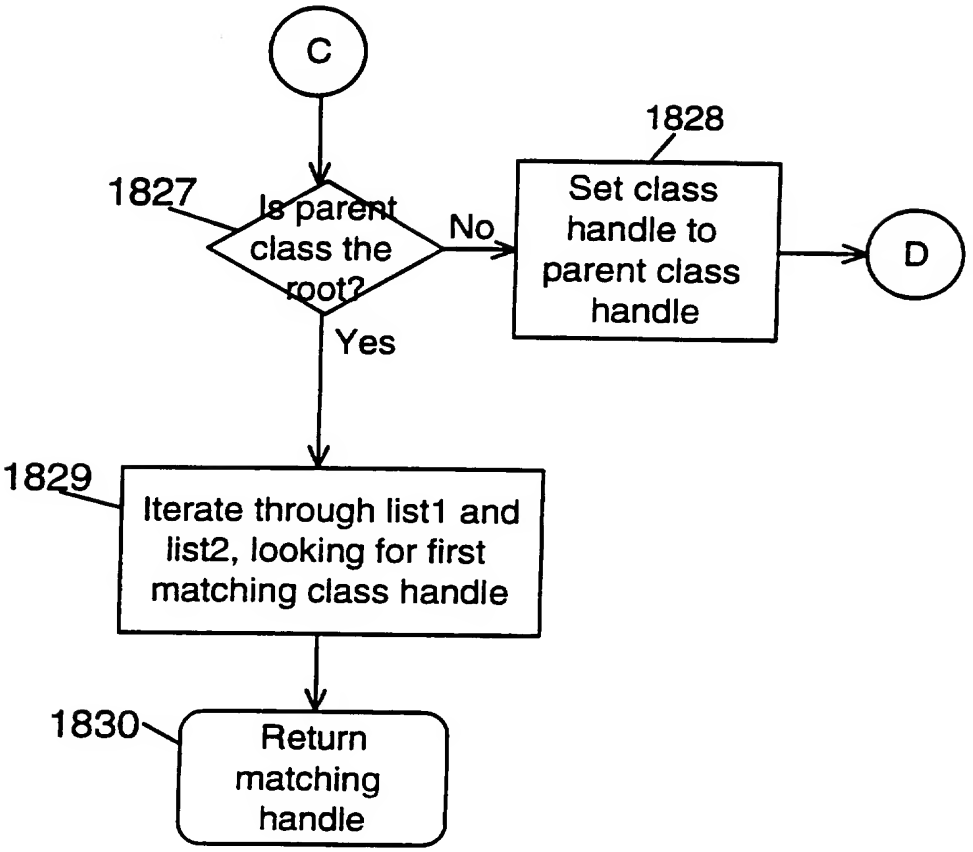

C
1828
1827
Is parent class the root?
No
Set class handle to parent class handle
D
Yes
1829
Iterate through list1 and list2, looking for first matching class handle
1830
Return matching handle

FIG. 80

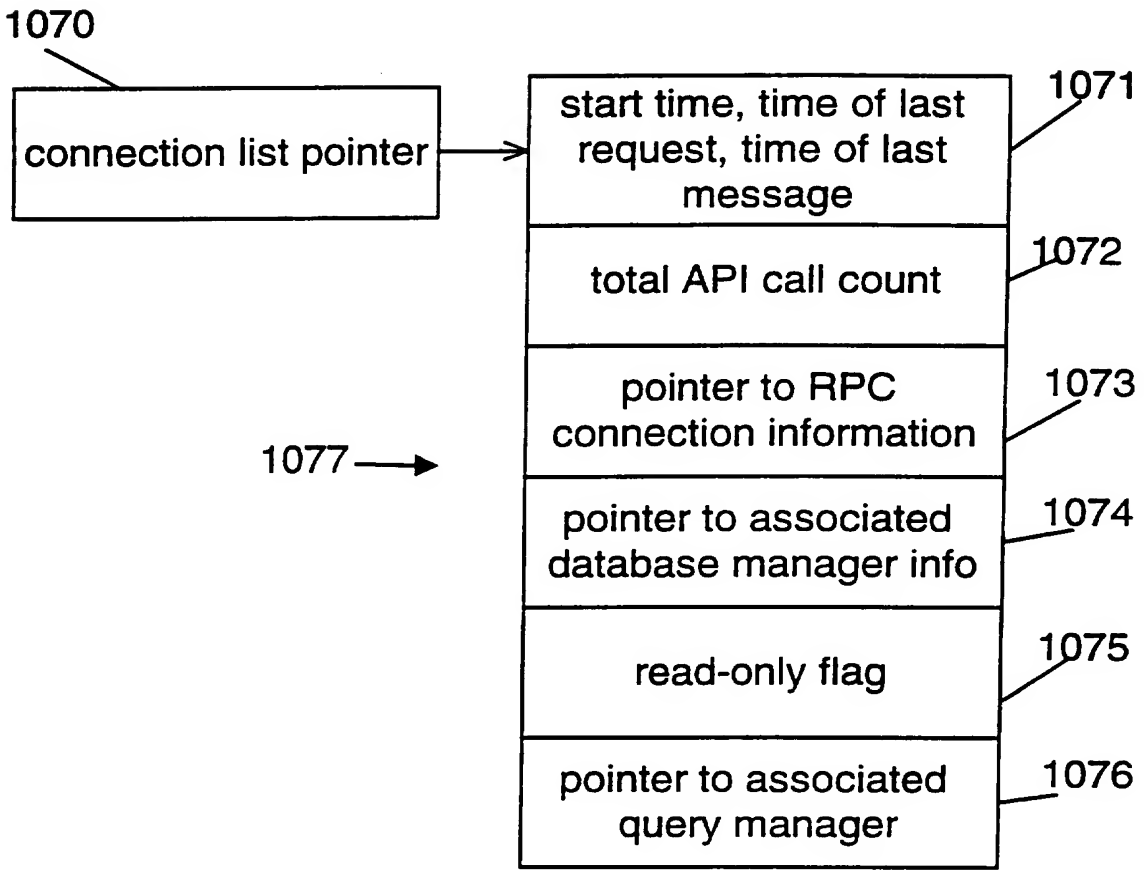

1070
connection list pointer
start time, time of last request, time of last message
1071
total API call count
1072
pointer to RPC connection information
1073
1077
pointer to associated database manager info
1074
read-only flag
1075
pointer to associated query manager
1076

FIG. 81

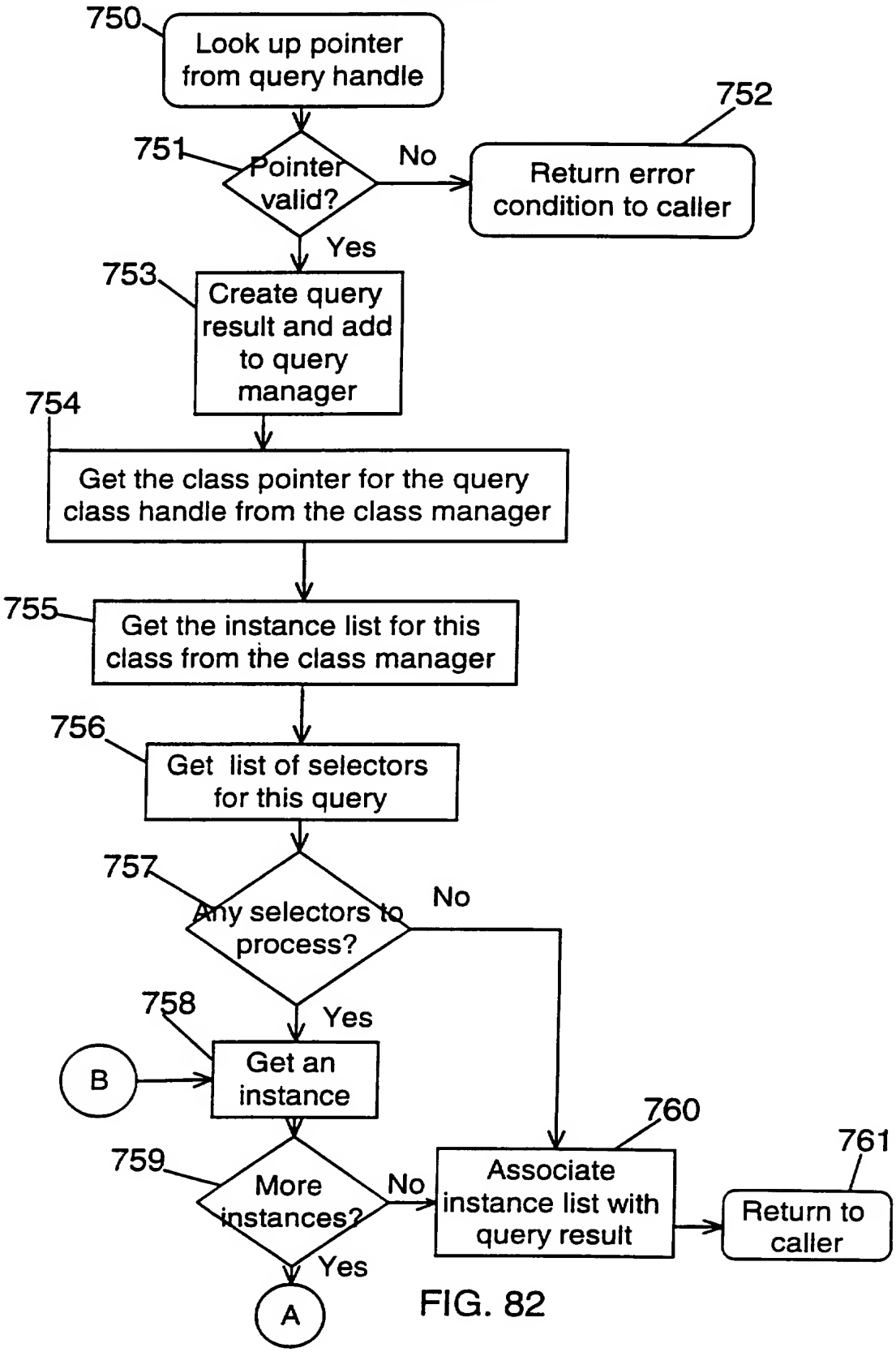

750
Look up pointer from query handle
751
Pointer valid?
No
752
Return error condition to caller
Yes
753
Create query result and add to query manager
754
Get the class pointer for the query class handle from the class manager
755
Get the instance list for this class from the class manager
756
Get list of selectors for this query
757
Any selectors to process?
No
Yes
758
B
Get an instance
759
More instances?
No
Yes
A
760
Associate instance list with query result
761
Return to caller

FIG. 82

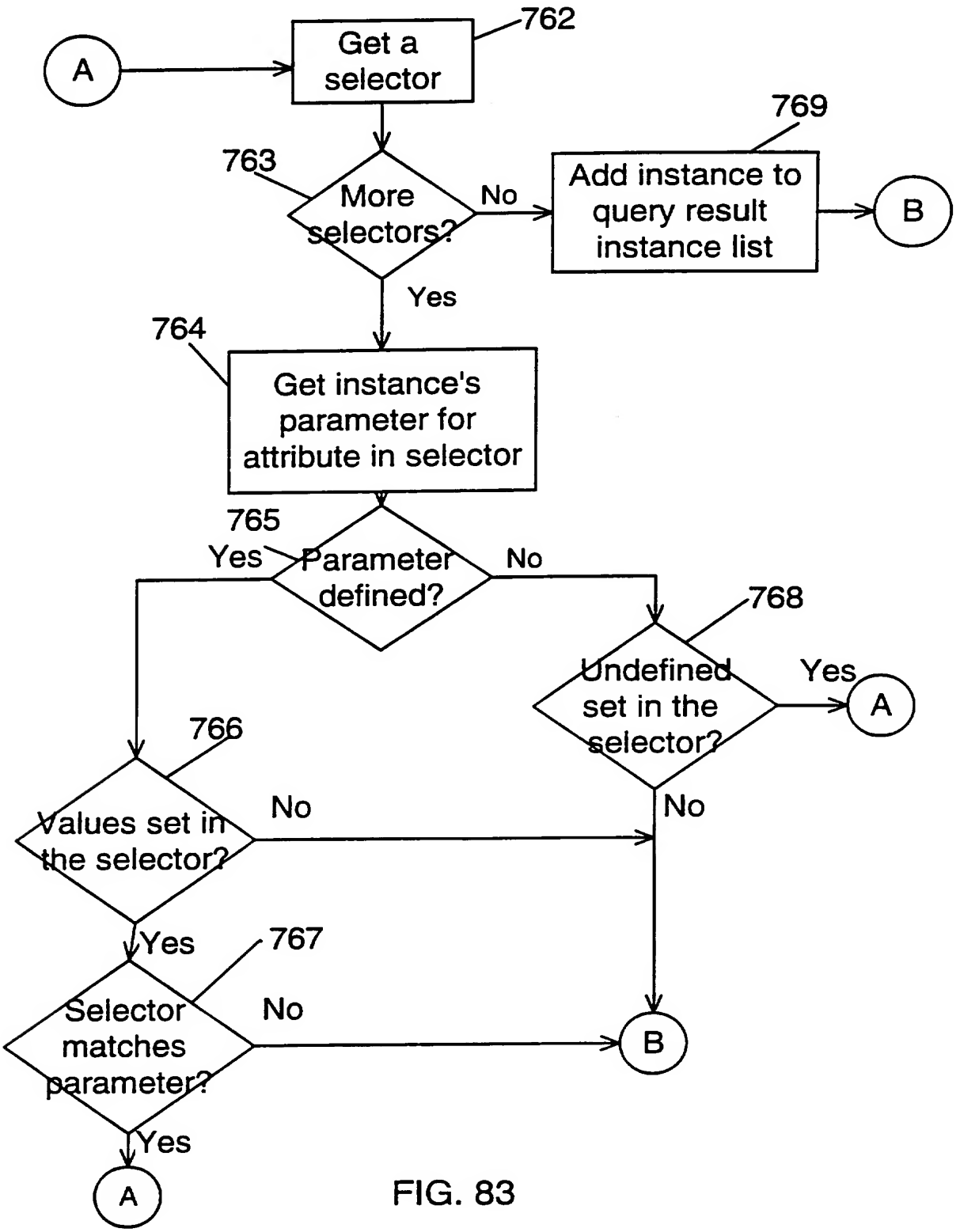

A
Get a selector
762
763
More selectors?
No
769
Add instance to query result instance list
B
Yes
764
Get instance's parameter for attribute in selector
765
Yes
Parameter defined?
No
768
Undefined set in the selector?
Yes
A
766
Values set in the selector?
No
No
Yes
767
Selector matches parameter?
No
B
Yes
A

FIG. 83

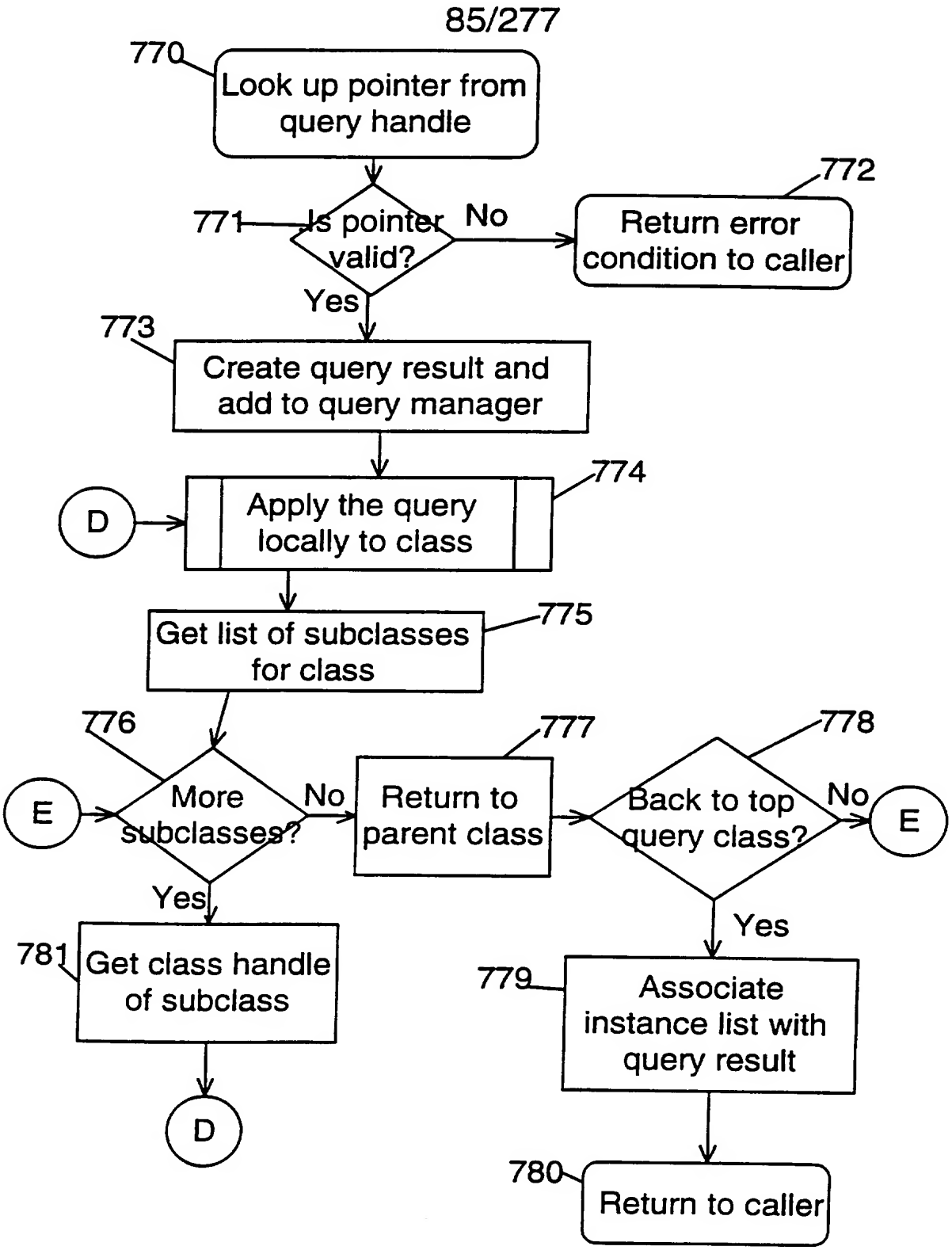

770
Look up pointer from query handle
771
Is pointer valid?
No
772
Return error condition to caller
Yes
773
Create query result and add to query manager
D
774
Apply the query locally to class
775
Get list of subclasses for class
776
E
More subclasses?
No
777
Return to parent class
778
Back to top query class?
No
E
Yes
781
Get class handle of subclass
D
Yes
779
Associate instance list with query result
780
Return to caller

FIG. 84

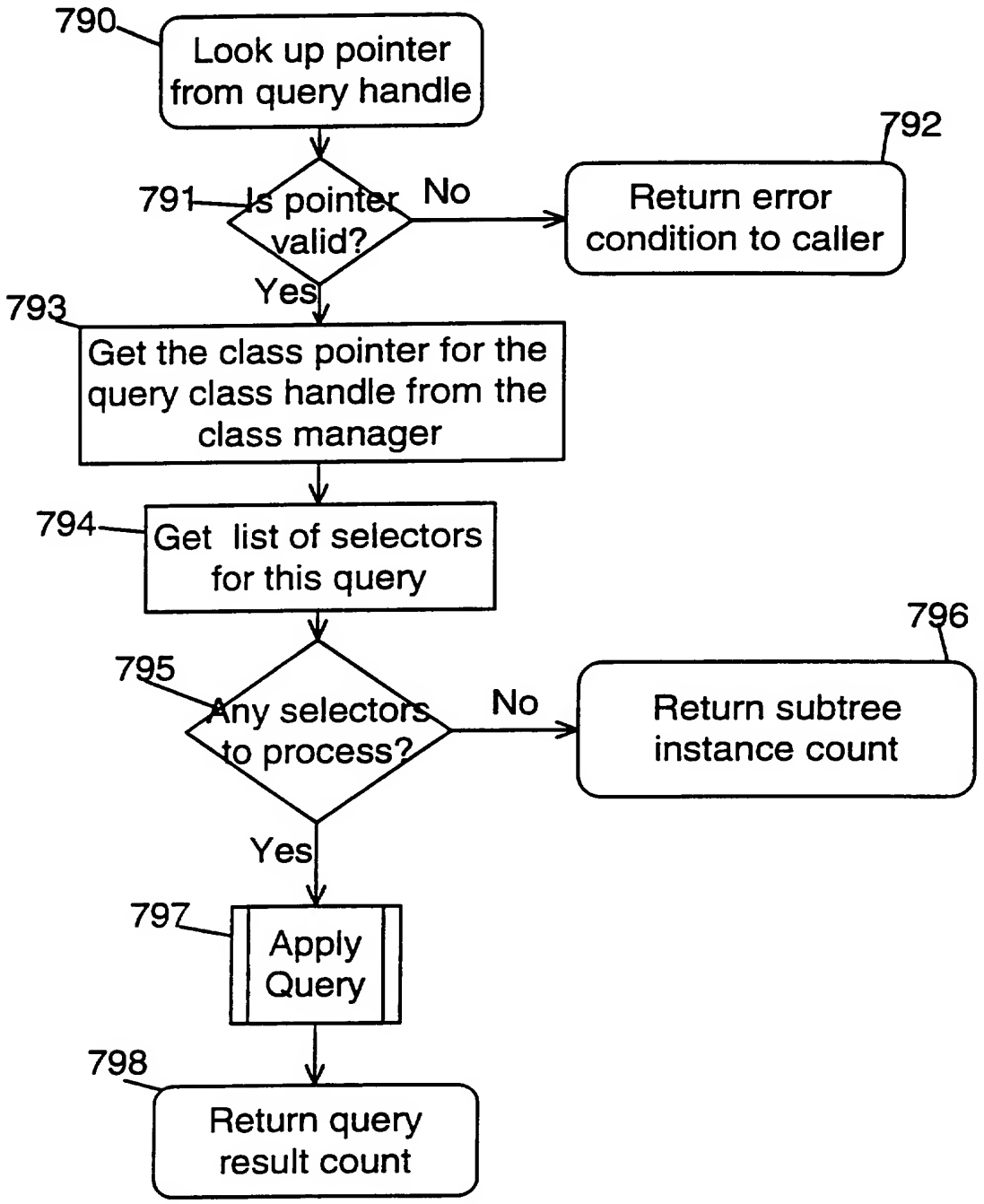

790
Look up pointer from query handle
791
Is pointer valid?
No
792
Return error condition to caller
Yes
793
Get the class pointer for the query class handle from the class manager
794
Get list of selectors for this query
795
Any selectors to process?
No
796
Return subtree instance count
Yes
797
Apply Query
798
Return query result count

FIG. 85

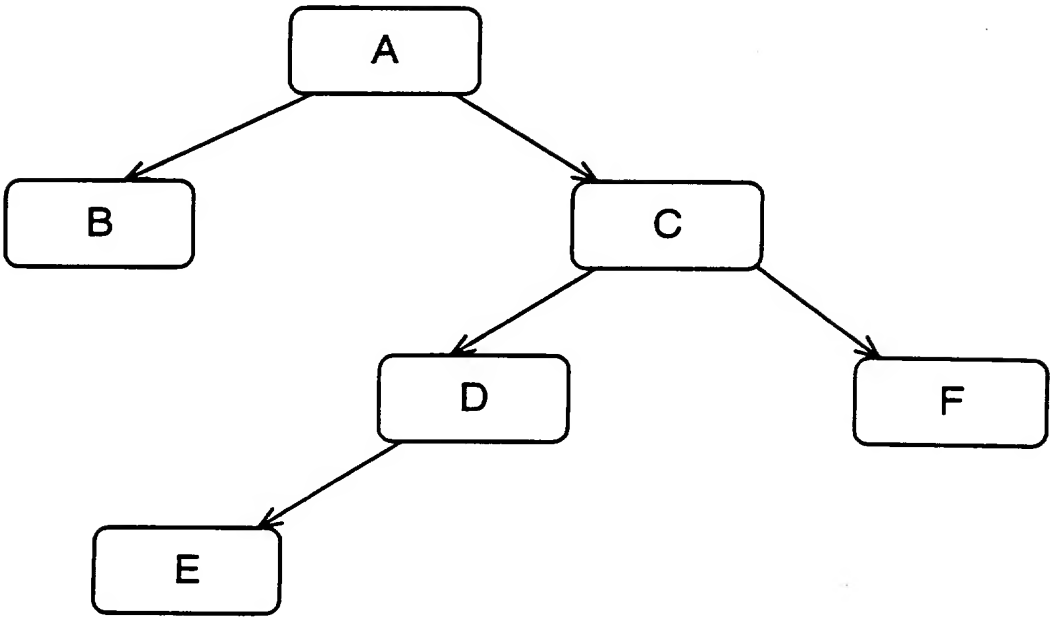


FIG. 86

Match Component	Component	Matched?
Base Number	2901	
Prefix	LM	No
Suffix	B	Yes
Manufacturer	AMD	No
# of Classes Found	1	Yes

FIG. 87

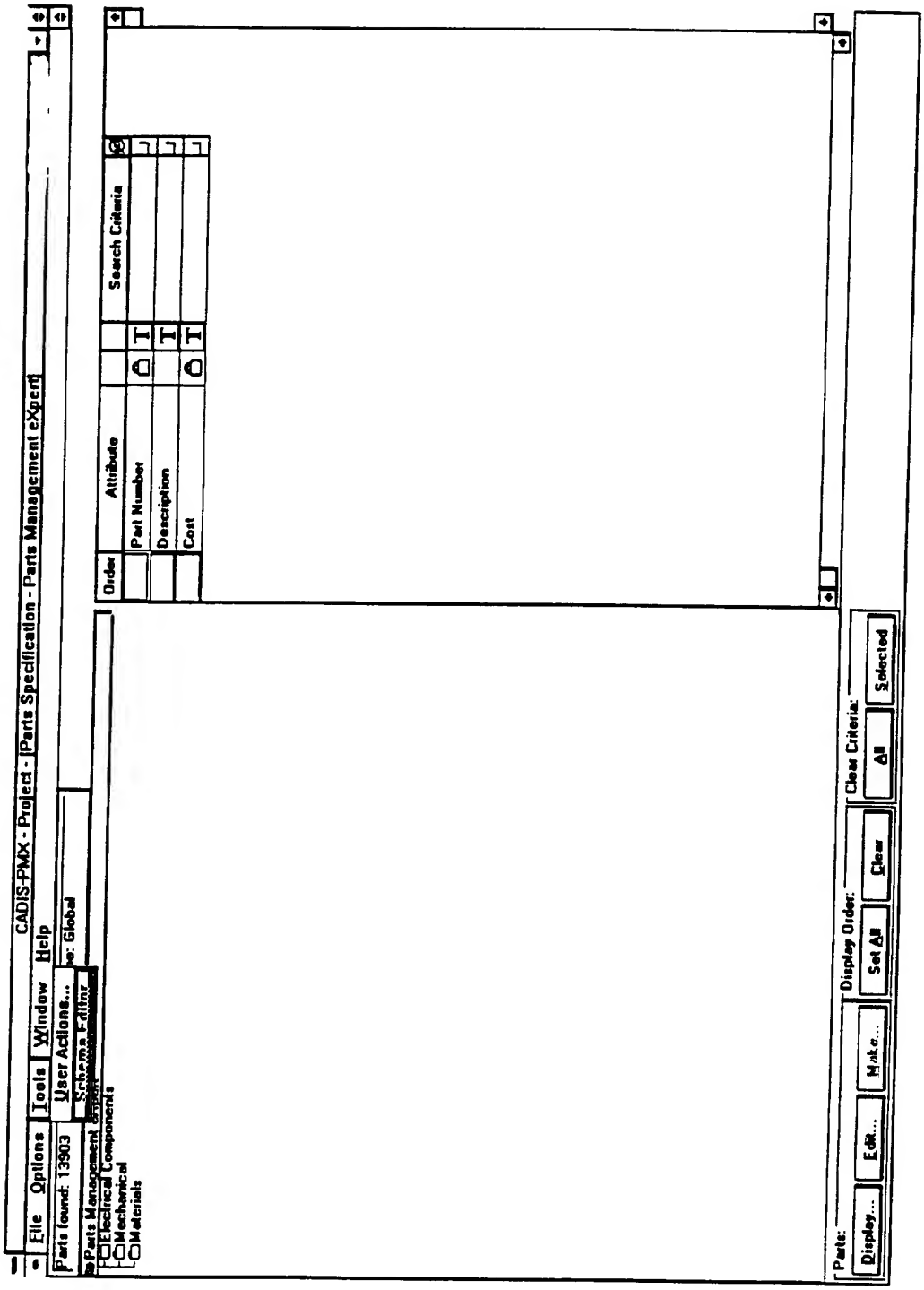

CADIS-PMX - Project - [Parts Specification - Parts Management eXpert]
File Options Tools Window Help
Parts found: 13903
User Actions...
Schema Editor
Parts Management Expert
Electrical Components
Mechanical
Materials
Order
Attribute
Search Criteria
Part Number
Description
Cost
Parts:
Display...
Edit...
Make...
Display Order:
Set All
Clear
Clear Criteria:
All
Selected

FIG. 88

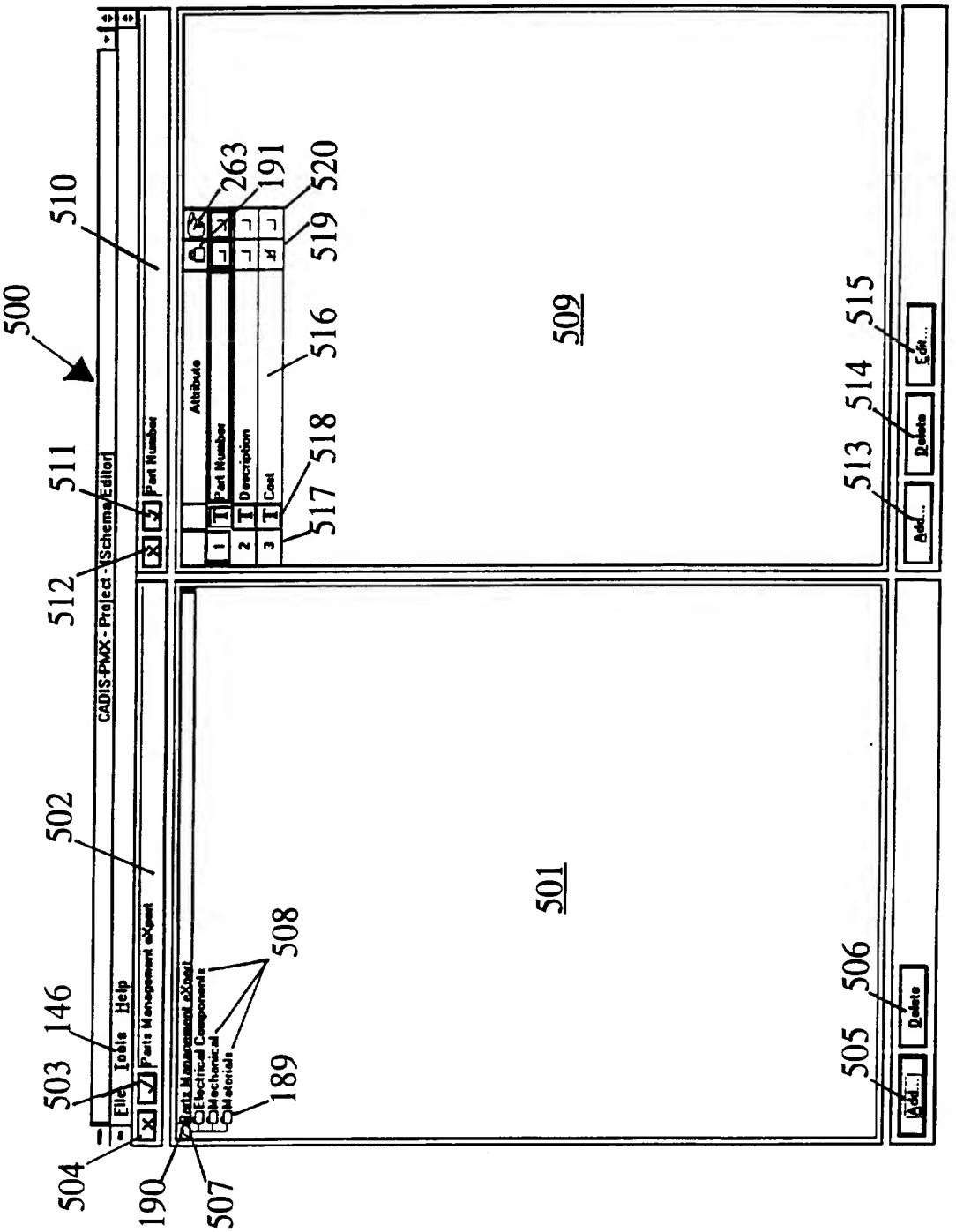

500
510
511
512
502
146
503
504
190
507
508
189
501
263
191
520
519
516
518
517
509
515
514
513
506
505
CADIS-PMX - Project - YSchema Editor
File Tools Help
Parts Management eXpert
Part Number
Attribute
Part Number
Description
Cost
Add...
Delete
Edit...
Electrical Components
Mechanical
Materials

FIG. 89

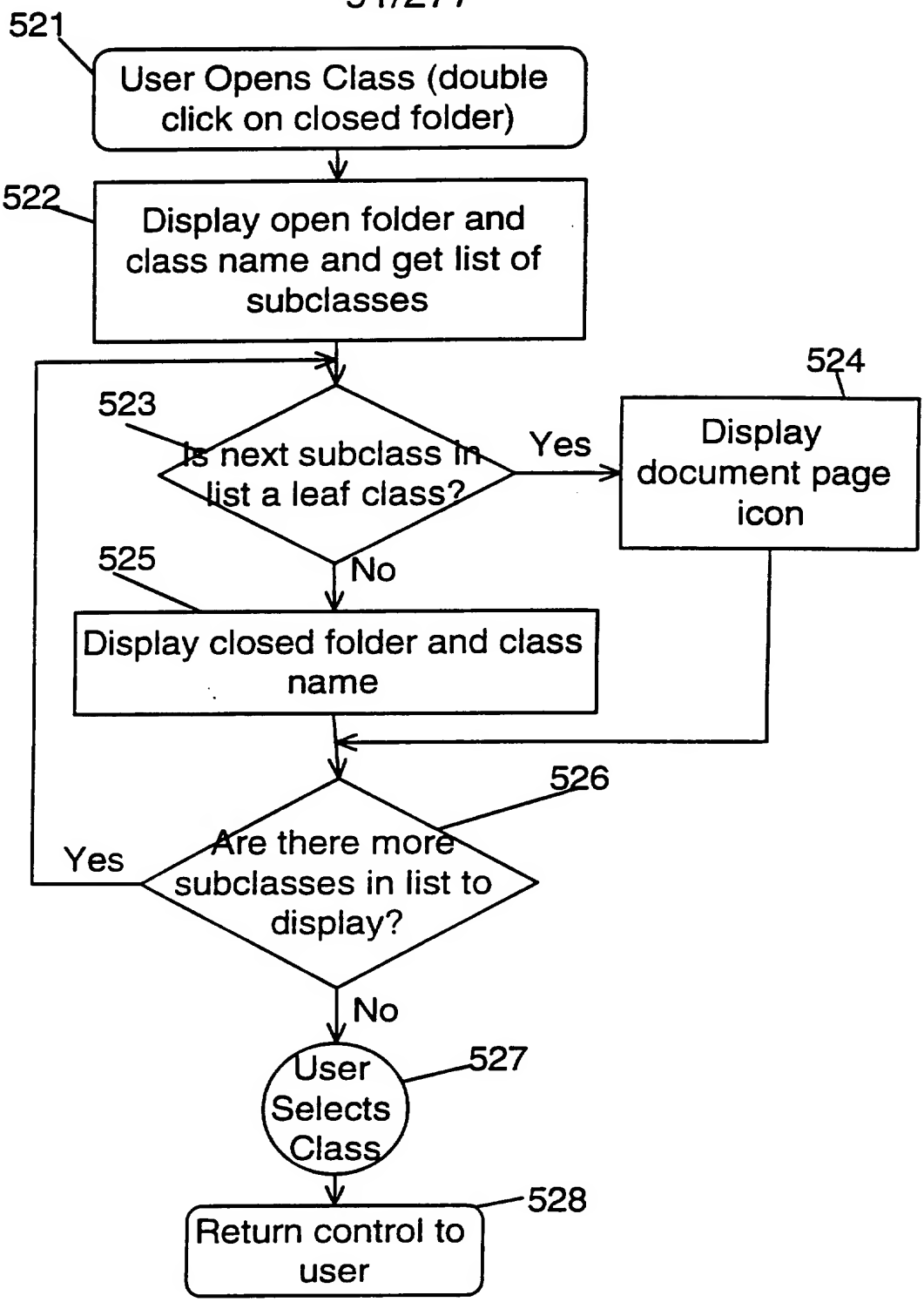

521
User Opens Class (double click on closed folder)
522
Display open folder and class name and get list of subclasses
523
Is next subclass in list a leaf class?
Yes
524
Display document page icon
No
525
Display closed folder and class name
526
Are there more subclasses in list to display?
Yes
No
User Selects Class
527
528
Return control to user

FIG. 90

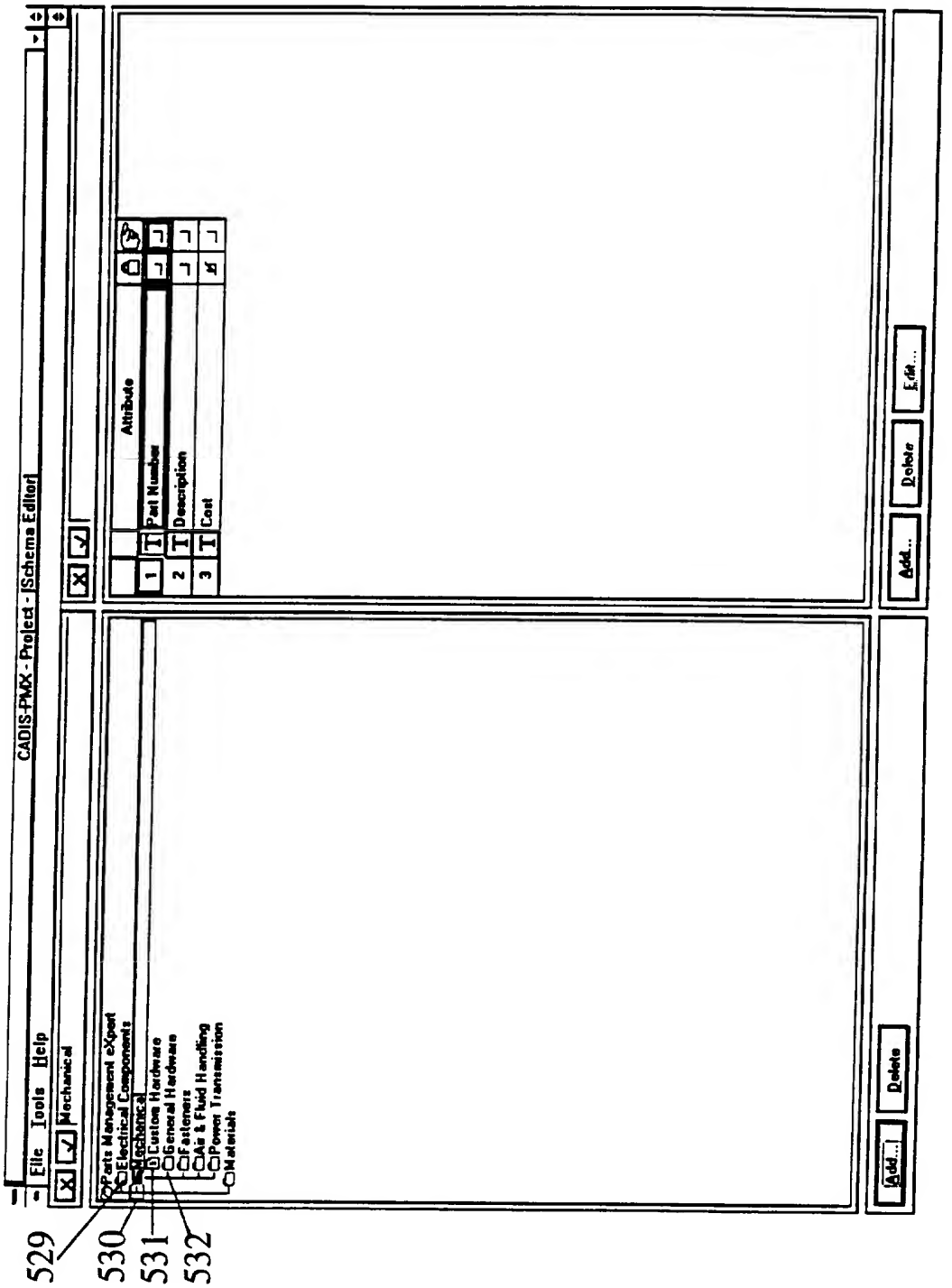

CADIS-PMX - Project - [Schema Editor]
File Tools Help
Mechanical
Parts Management eXpert
Electrical Components
Mechanical
Custom Hardware
General Hardware
Fasteners
Air & Fluid Handling
Power Transmission
Materials
Attribute
Part Number
Description
Cost
Add...
Delete
Edit...
529
530
531
532

FIG. 91

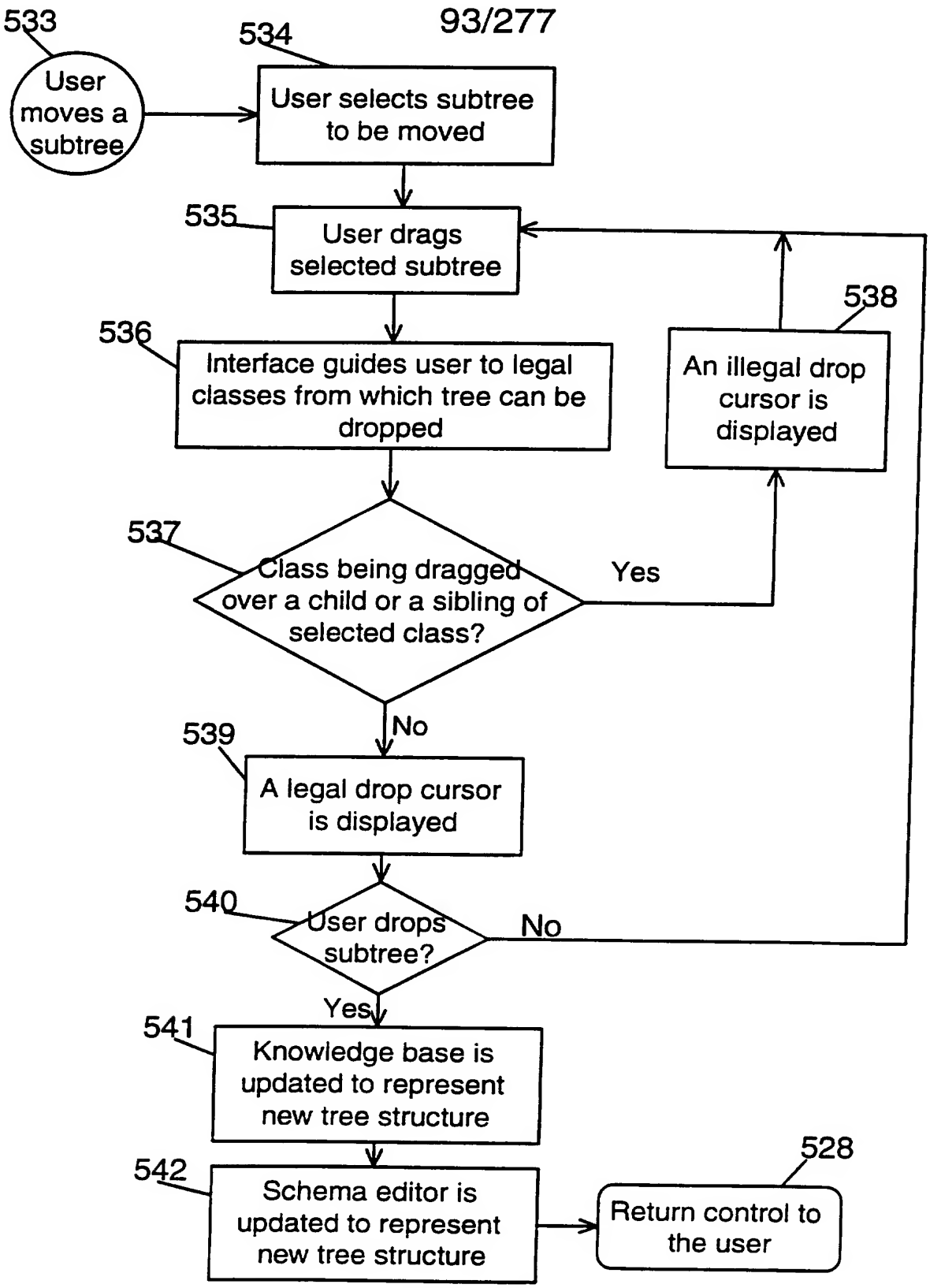

533
User moves a subtree
534
User selects subtree to be moved
535
User drags selected subtree
536
Interface guides user to legal classes from which tree can be dropped
537
Class being dragged over a child or a sibling of selected class?
Yes
538
An illegal drop cursor is displayed
No
539
A legal drop cursor is displayed
540
User drops subtree?
No
Yes
541
Knowledge base is updated to represent new tree structure
542
Schema editor is updated to represent new tree structure
528
Return control to the user

FIG. 92

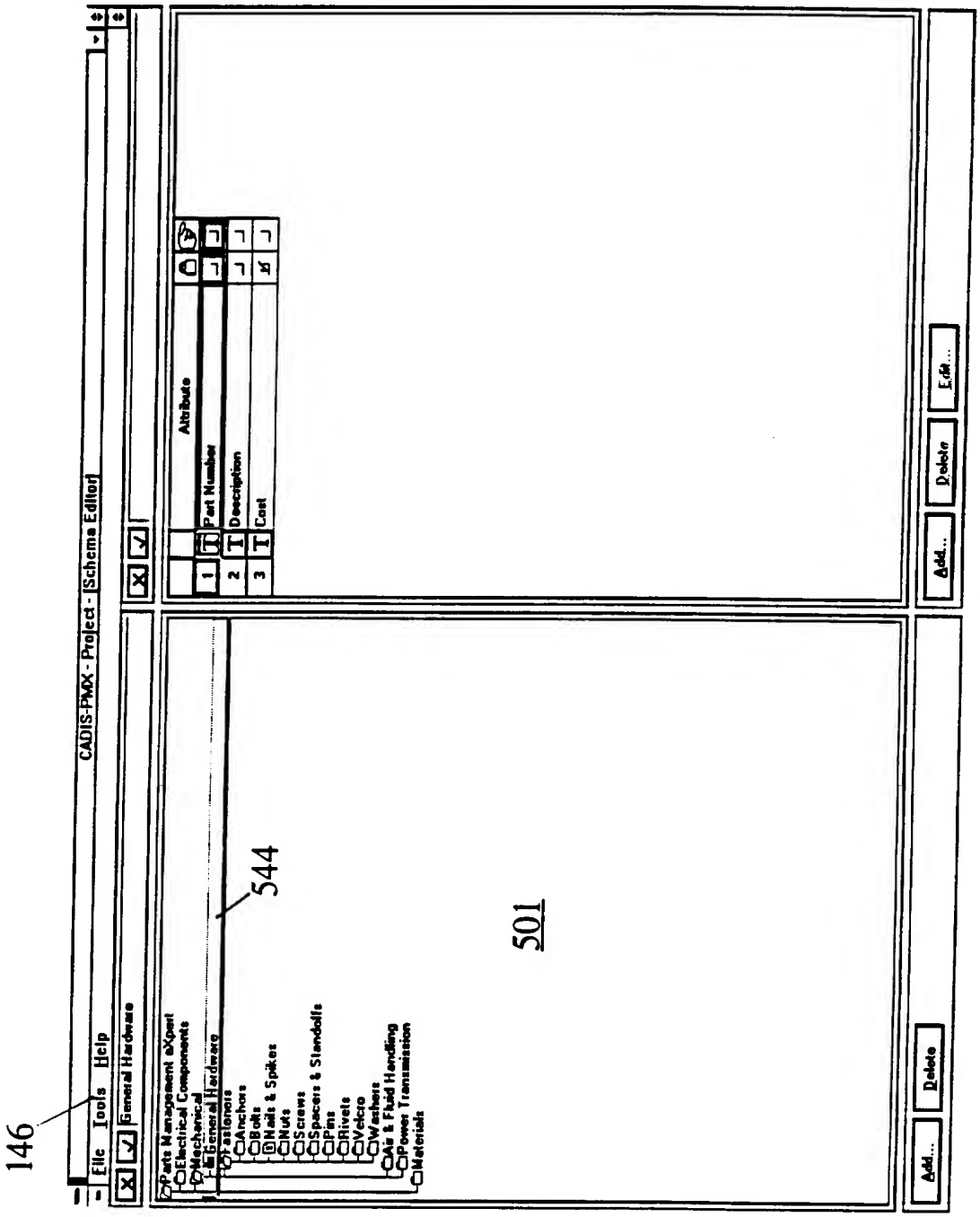

146
544
501
CADIS-PMX - Project - [Schema Editor]
File Tools Help
General Hardware
Parts Management eXpert
Electrical Components
Mechanical
General Hardware
Fasteners
Anchors
Bolts
Nails & Spikes
Nuts
Screws
Spacers & Standoffs
Pins
Rivets
Velcro
Washers
Air & Fluid Handling
Power Transmission
Materials
Attribute
Part Number
Description
Cost
Add...
Delete
Edit...

FIG. 93

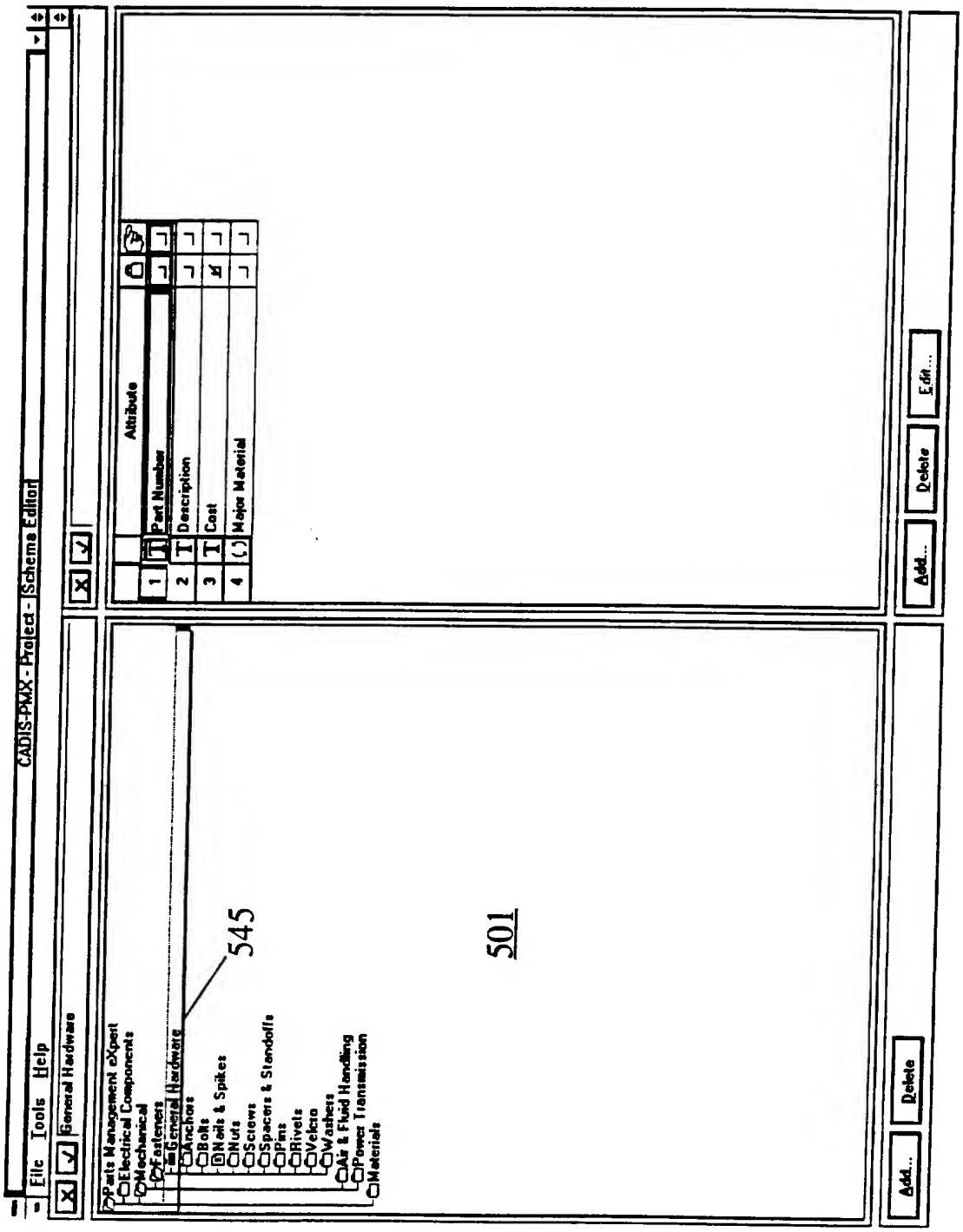

CADIS-PMX - Project - Schema Editor
File Tools Help
General Hardware
Parts Management eXpert
Electrical Components
Mechanical
Fasteners
General Hardware
Anchors
Bolts
Nails & Spikes
Nuts
Screws
Spacers & Standoffs
Pins
Rivets
Velcro
Washers
Air & Fluid Handling
Power Transmission
Materials
545
501
Attribute
Part Number
Description
Cost
Major Material
Add...
Delete
Edit...

FIG. 94

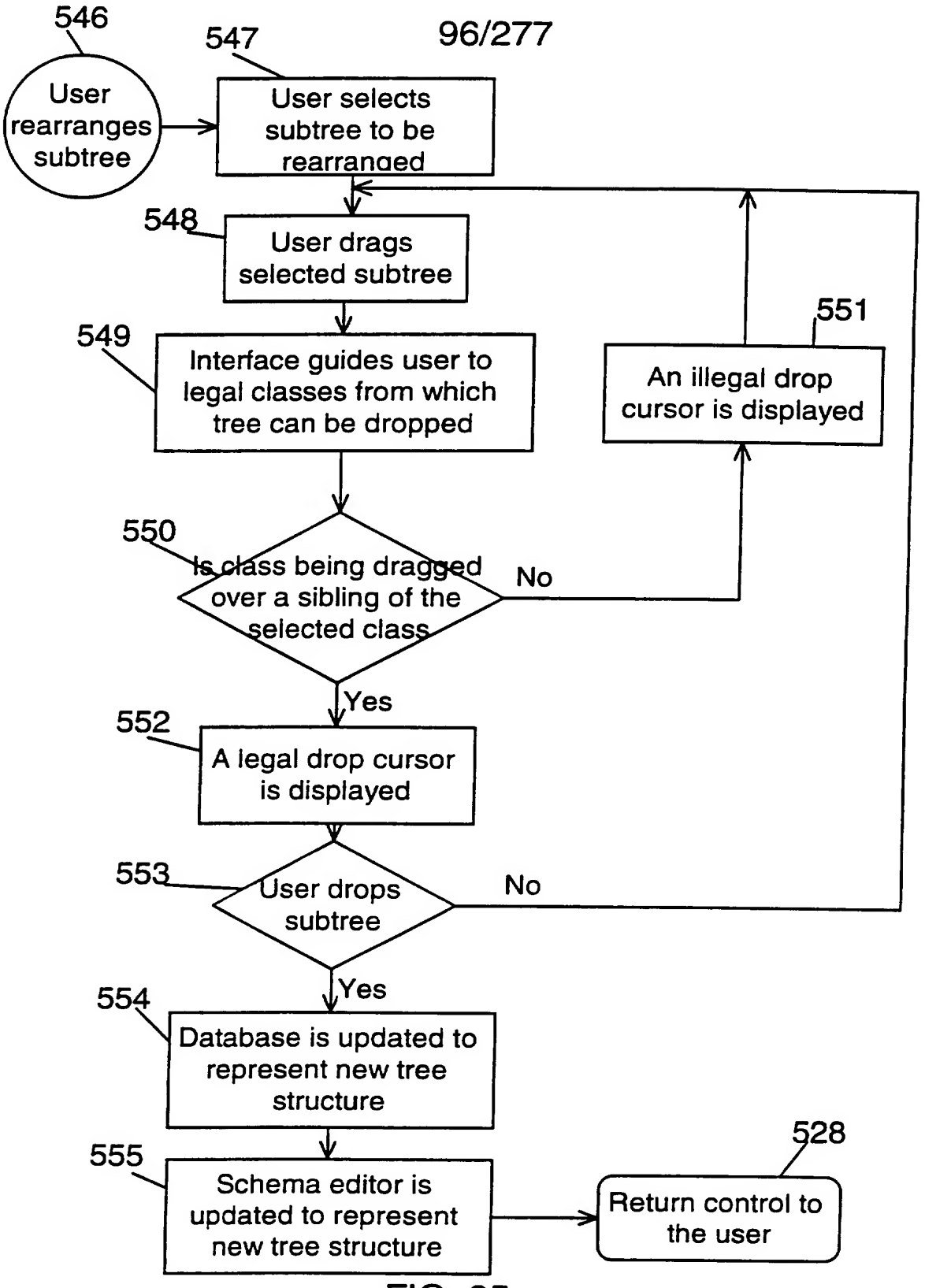

546
User rearranges subtree
547
User selects subtree to be rearranged
548
User drags selected subtree
549
Interface guides user to legal classes from which tree can be dropped
550
Is class being dragged over a sibling of the selected class
No
551
An illegal drop cursor is displayed
Yes
552
A legal drop cursor is displayed
553
User drops subtree
No
Yes
554
Database is updated to represent new tree structure
555
Schema editor is updated to represent new tree structure
528
Return control to the user

FIG. 95

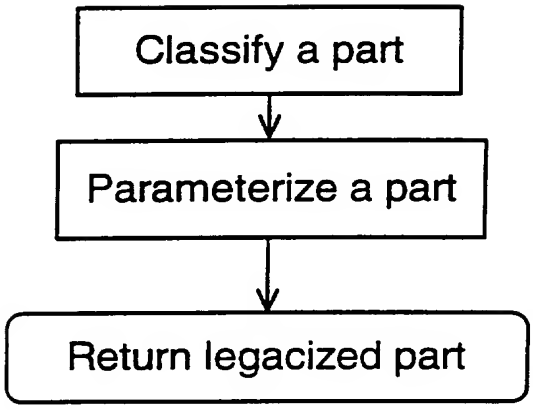


FIG. 96

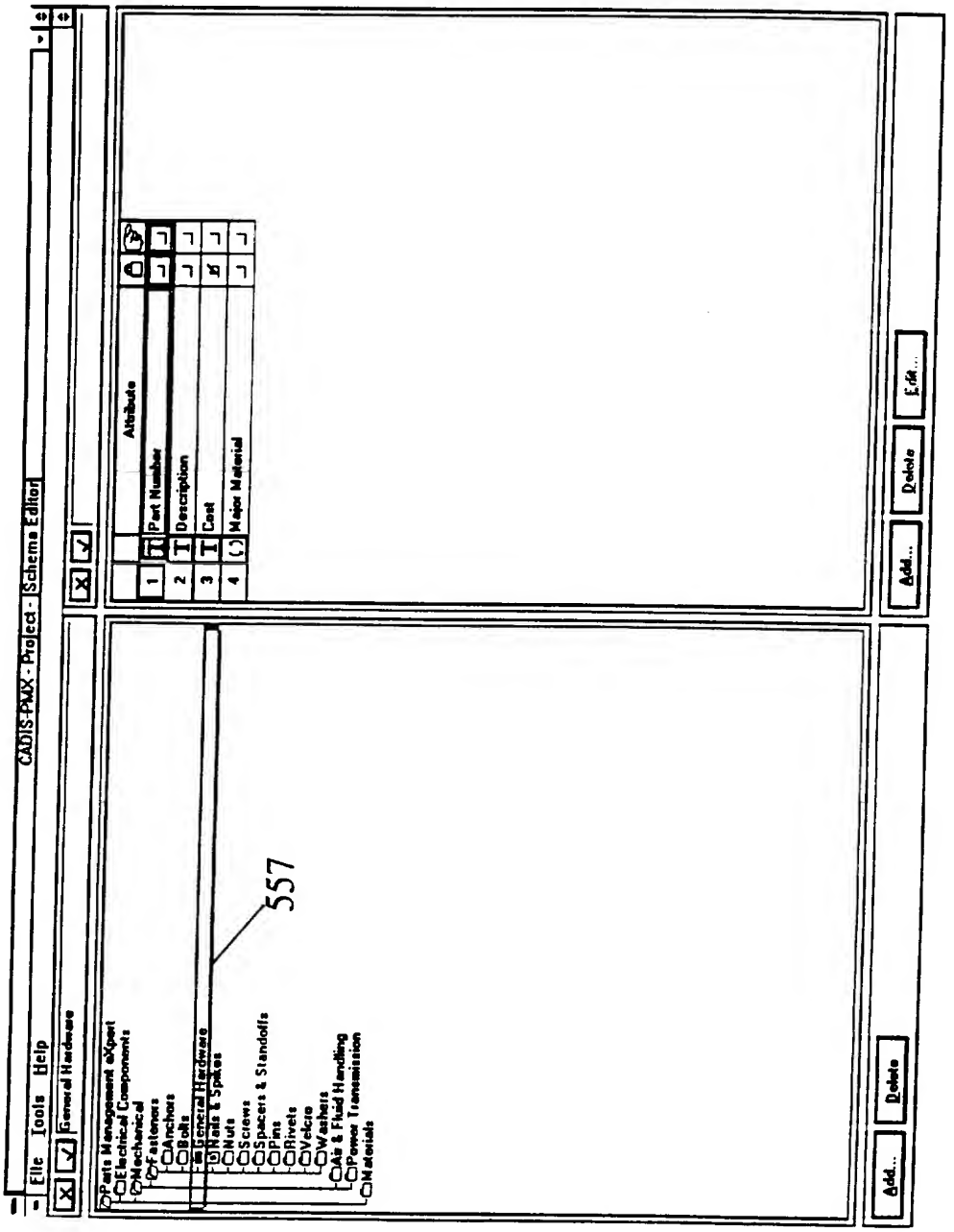

CADIS-PMX - Project - Schema Editor
File Tools Help
General Hardware
Parts Management eXpert
Electrical Components
Mechanical
Fasteners
Anchors
Bolts
General Hardware
Nails & Spikes
Nuts
Screws
Spacers & Standoffs
Pins
Rivets
Velcro
Washers
Air & Fluid Handling
Power Transmission
Materials
Attribute
Part Number
Description
Cost
Major Material
Add...
Delete
Edit...
557

FIG. 97

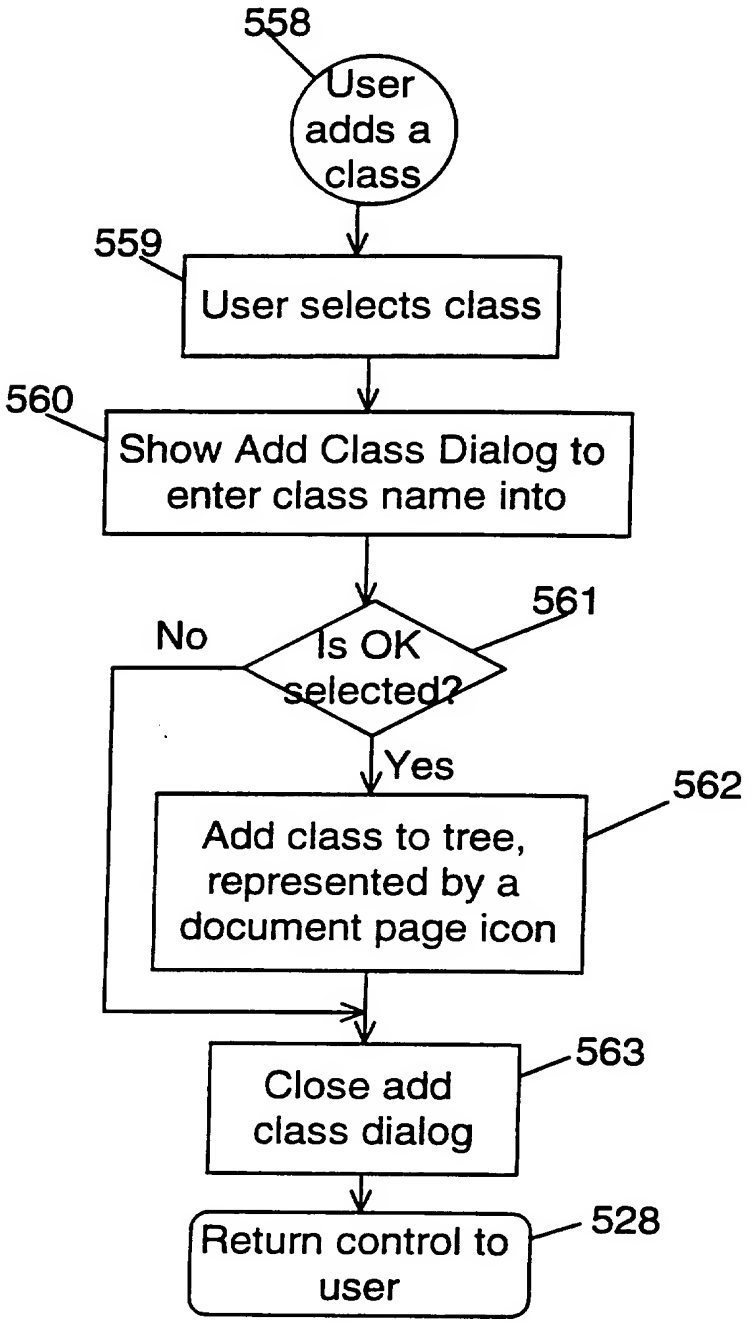

558
User adds a class
559
User selects class
560
Show Add Class Dialog to enter class name into
561
No
Is OK selected?
Yes
562
Add class to tree, represented by a document page icon
563
Close add class dialog
528
Return control to user

FIG. 98

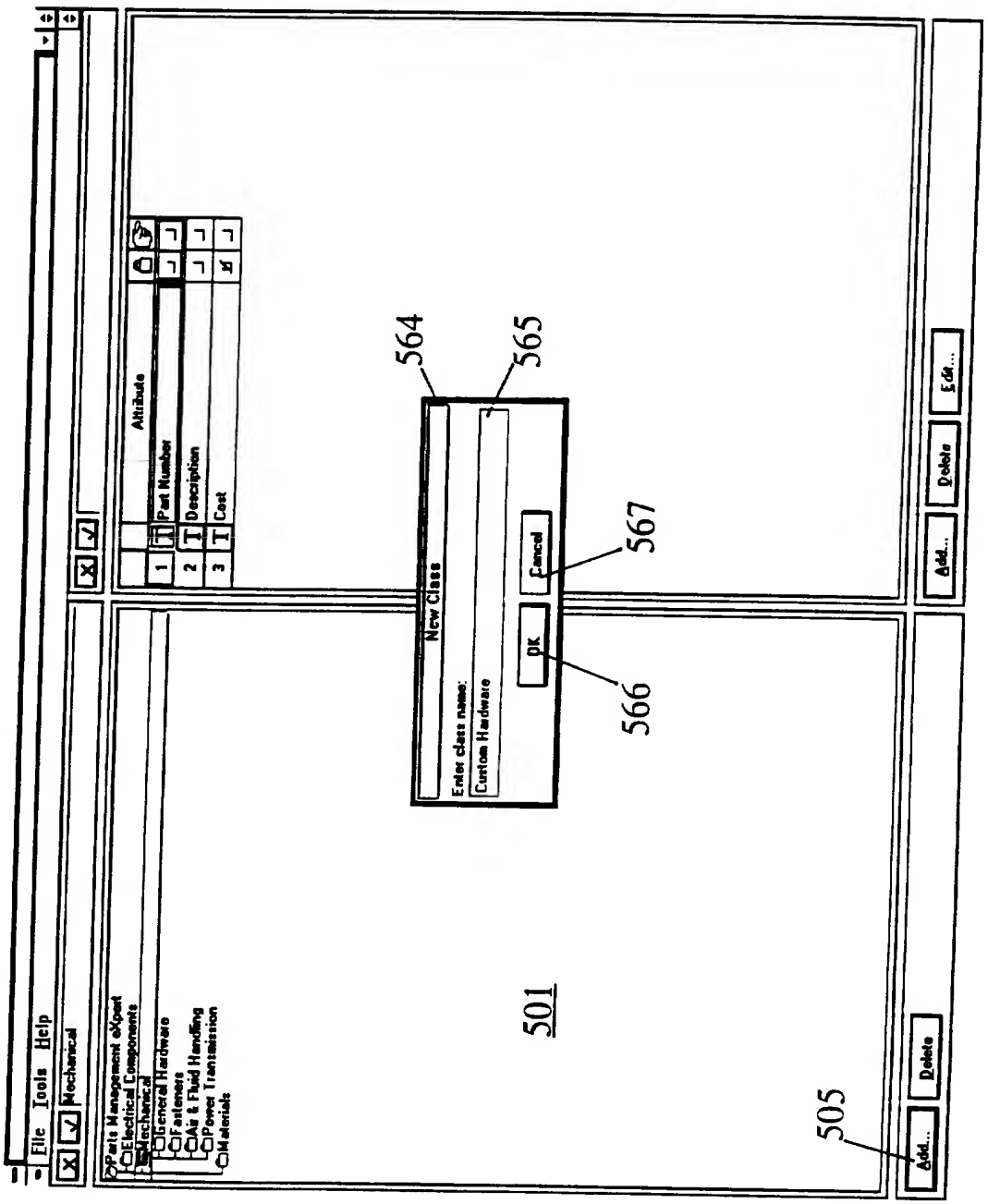

564
565
567
566
501
505
File Tools Help
Mechanical
Parts Management eXpert
Electrical Components
Mechanical
General Hardware
Fasteners
Air & Fluid Handling
Power Transmission
Materials
Attribute
Part Number
Description
Cost
New Class
Enter class name:
Custom Hardware
OK
Cancel
Add...
Delete
Edit...

FIG. 99

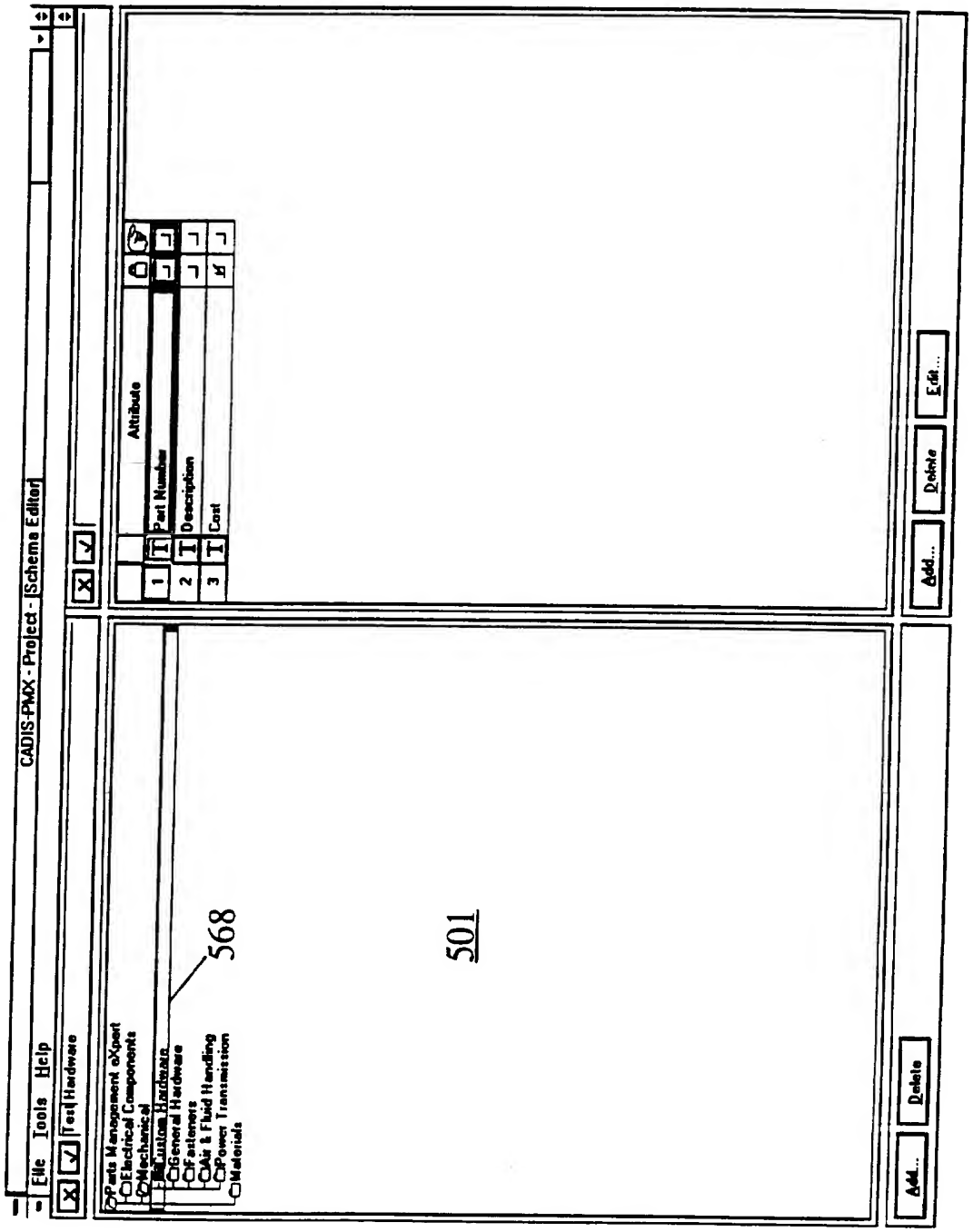

CADIS-PMX - Project - [Schema Editor]
File Tools Help
Test Hardware
Parts Management eXpert
Electrical Components
Mechanical
Custom Hardware
General Hardware
Fasteners
Air & Fluid Handling
Power Transmission
Materials
568
501
Attribute
1 Part Number
2 Description
3 Cost
Add...
Delete
Edit...

FIG. 100

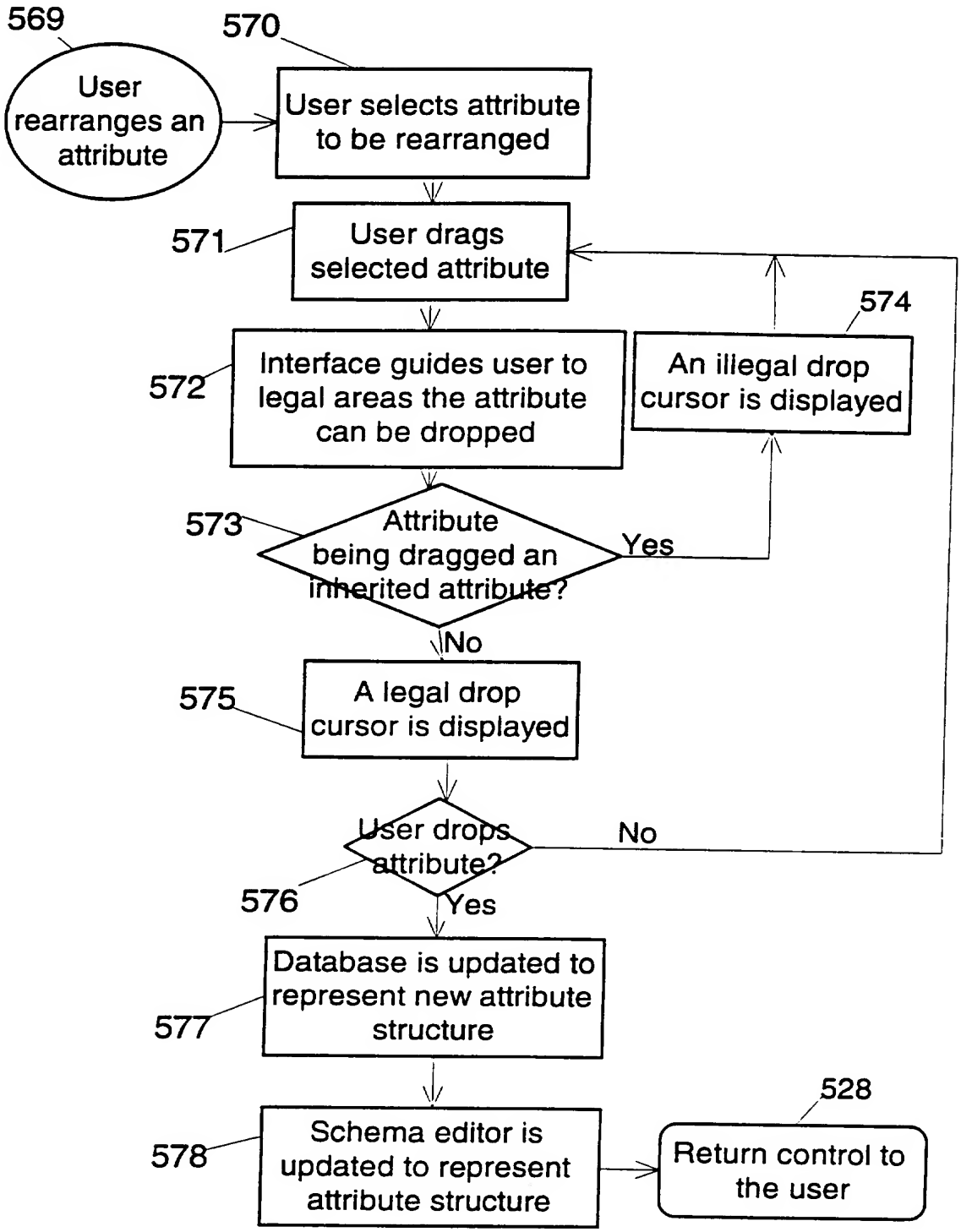

569
User rearranges an attribute
570
User selects attribute to be rearranged
571
User drags selected attribute
572
Interface guides user to legal areas the attribute can be dropped
574
An illegal drop cursor is displayed
573
Attribute being dragged an inherited attribute?
Yes
No
575
A legal drop cursor is displayed
User drops attribute?
No
576
Yes
Database is updated to represent new attribute structure
577
528
578
Schema editor is updated to represent attribute structure
Return control to the user

FIG. 101

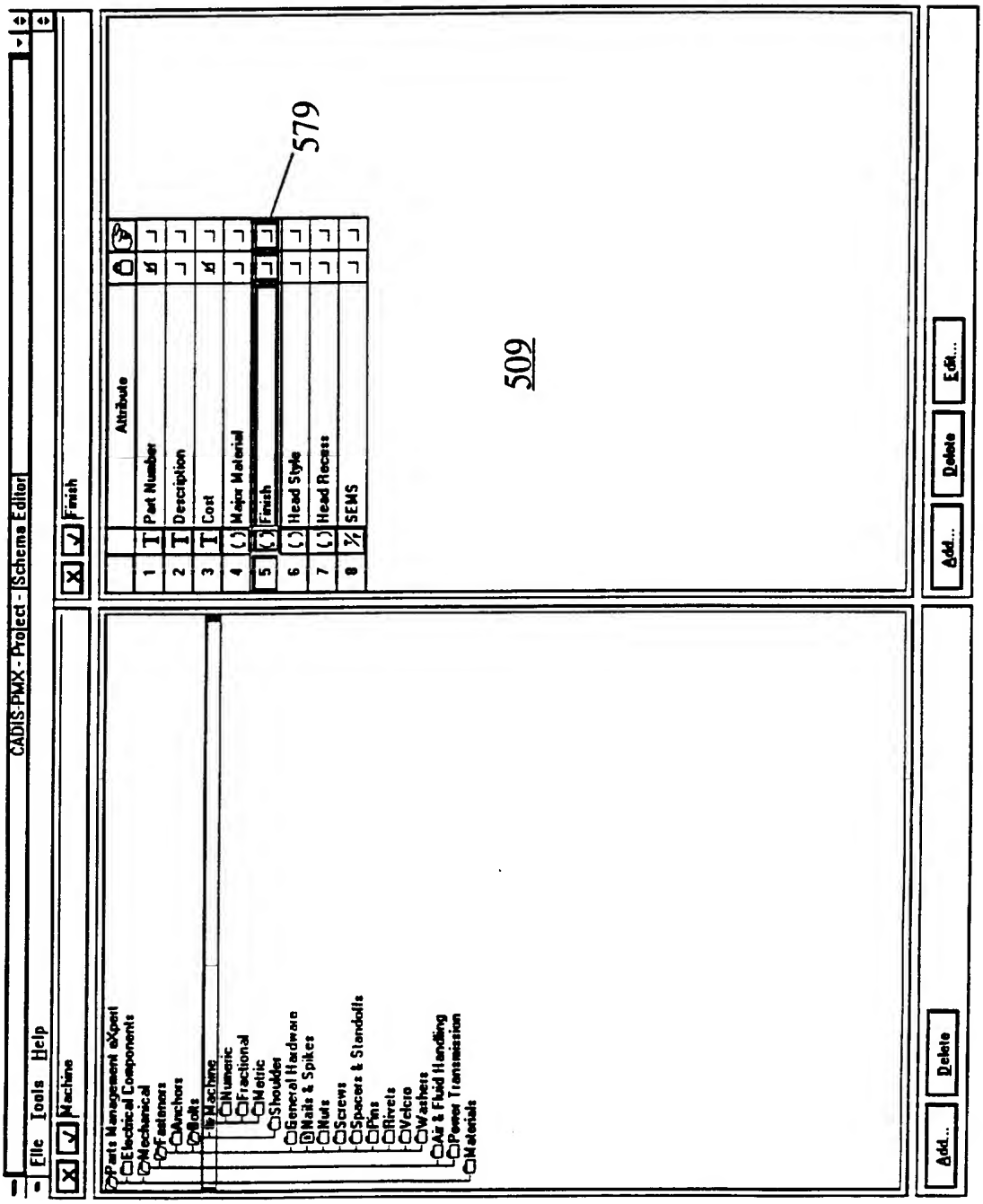

CADIS-PMX - Project - Schema Editor
File Tools Help
Machine
Finish
Parts Management eXpert
Electrical Components
Mechanical
Fasteners
Anchors
Bolts
Machine
Numeric
Fractional
Metric
Shoulder
General Hardware
Nails & Spikes
Nuts
Screws
Spacers & Standoffs
Pins
Rivets
Velcro
Washers
Air & Fluid Handling
Power Transmission
Materials
Attribute
Part Number
Description
Cost
Major Material
Finish
Head Style
Head Recess
SEMS
579
509
Add...
Delete
Edit...

FIG. 102

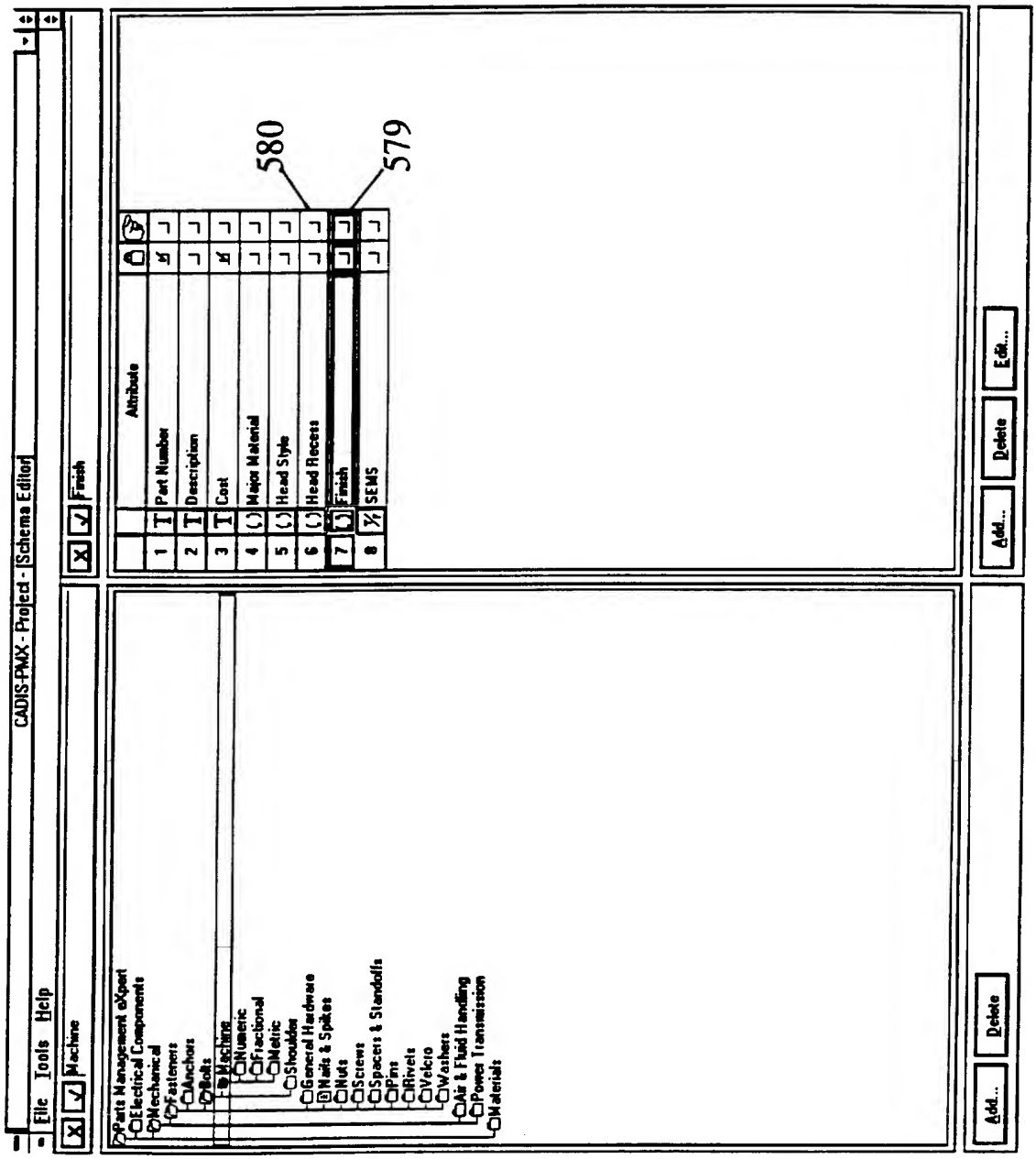

CADIS-PMX - Project - [Schema Editor]
File Tools Help
Machine
Finish
Parts Management eXpert
Electrical Components
Mechanical
Fasteners
Anchors
Bolts
Machine
Numeric
Fractional
Metric
Shoulder
General Hardware
Nails & Spikes
Nuts
Screws
Spacers & Standoffs
Pins
Rivets
Velcro
Washers
Air & Fluid Handling
Power Transmission
Materials
Attribute
1 Part Number
2 Description
3 Cost
4 Major Material
5 Head Style
6 Head Recess
7 Finish
8 SEMS
580
579
Add...
Delete
Edit...

FIG. 103

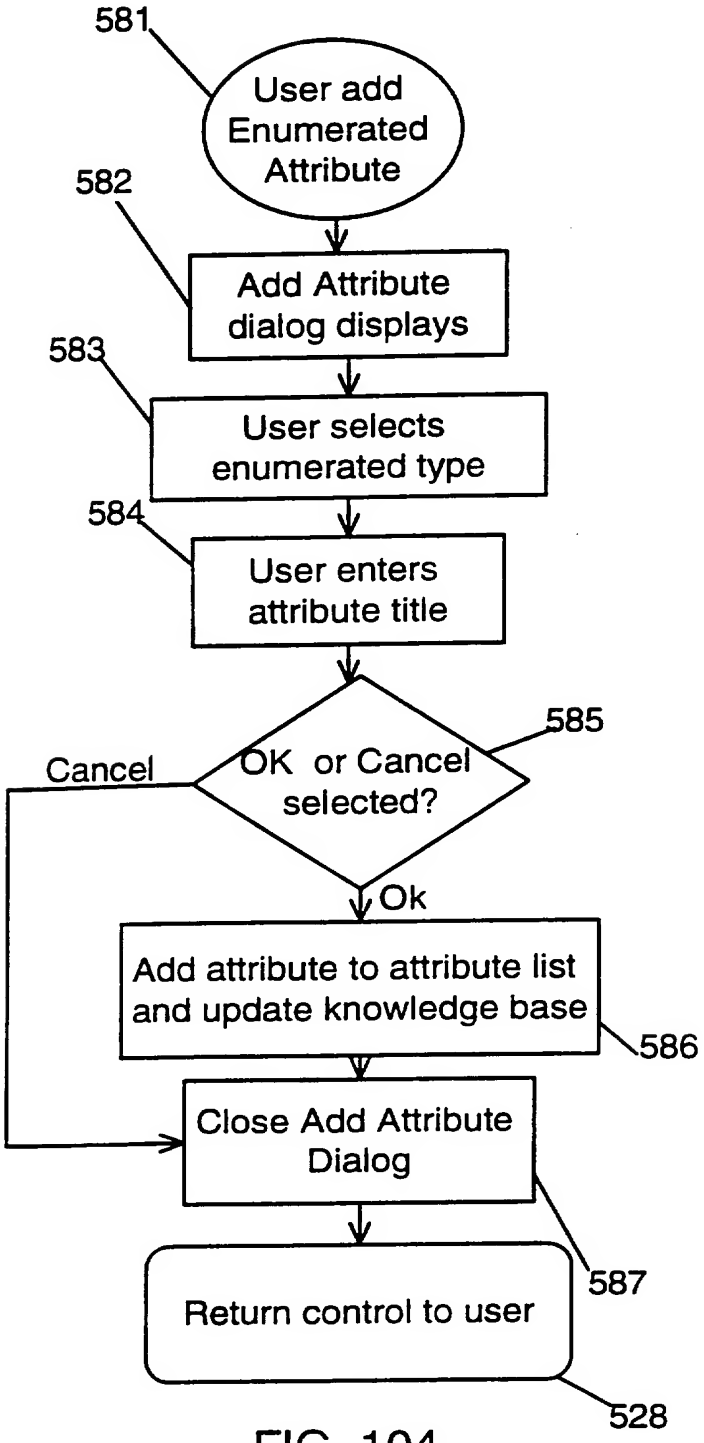

581
User add Enumerated Attribute
582
Add Attribute dialog displays
583
User selects enumerated type
584
User enters attribute title
585
OK or Cancel selected?
Cancel
Ok
Add attribute to attribute list and update knowledge base
586
Close Add Attribute Dialog
587
Return control to user
528

FIG. 104

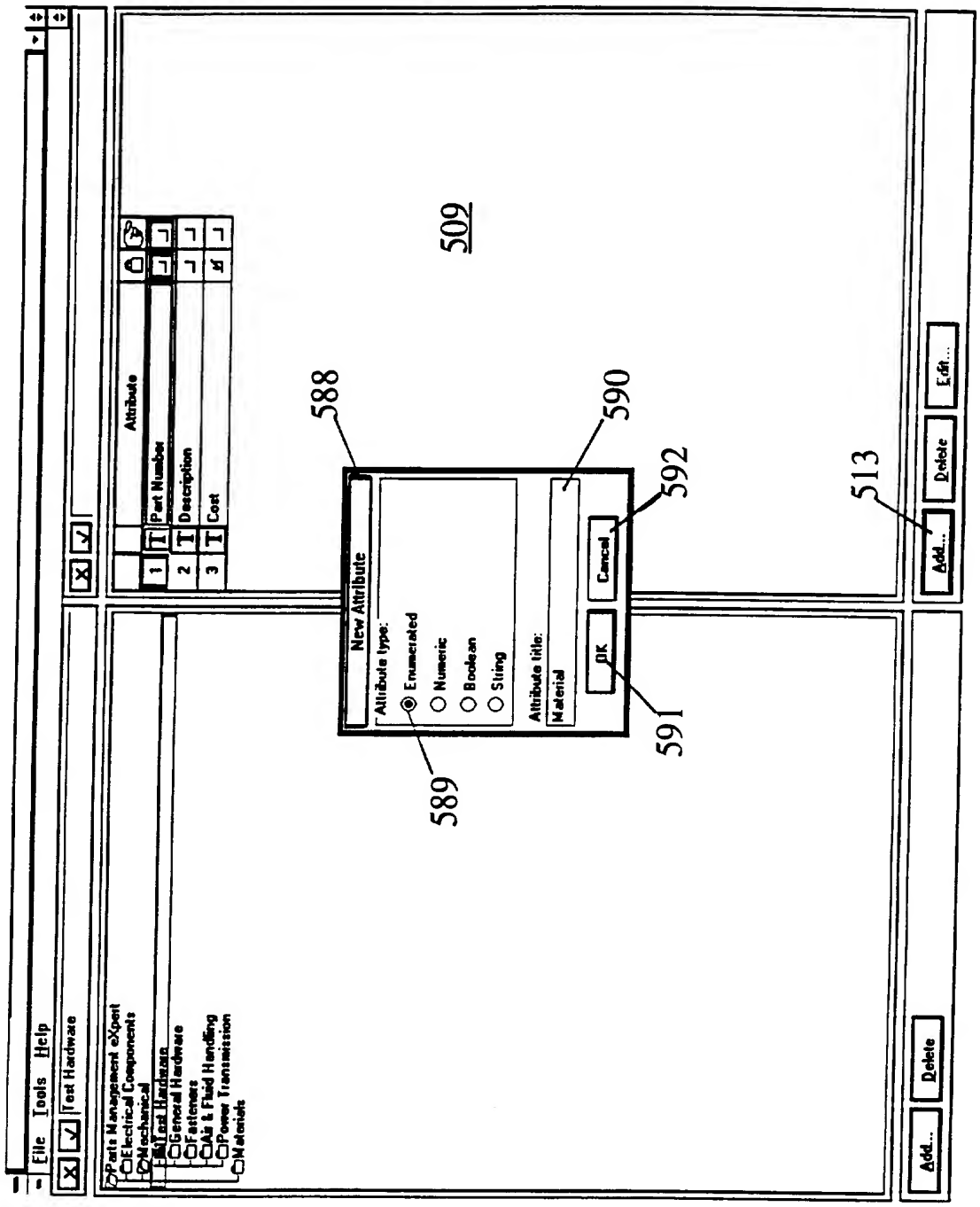

509
588
590
592
513
589
591
New Attribute
Attribute type:
Enumerated
Numeric
Boolean
String
Attribute title:
Material
OK
Cancel
Attribute
Part Number
Description
Cost
Add...
Delete
Edit...
File Tools Help
Test Hardware
Parts Management eXpert
Electrical Components
Mechanical
Test Hardware
General Hardware
Fasteners
Air & Fluid Handling
Power Transmission
Materials

FIG. 105

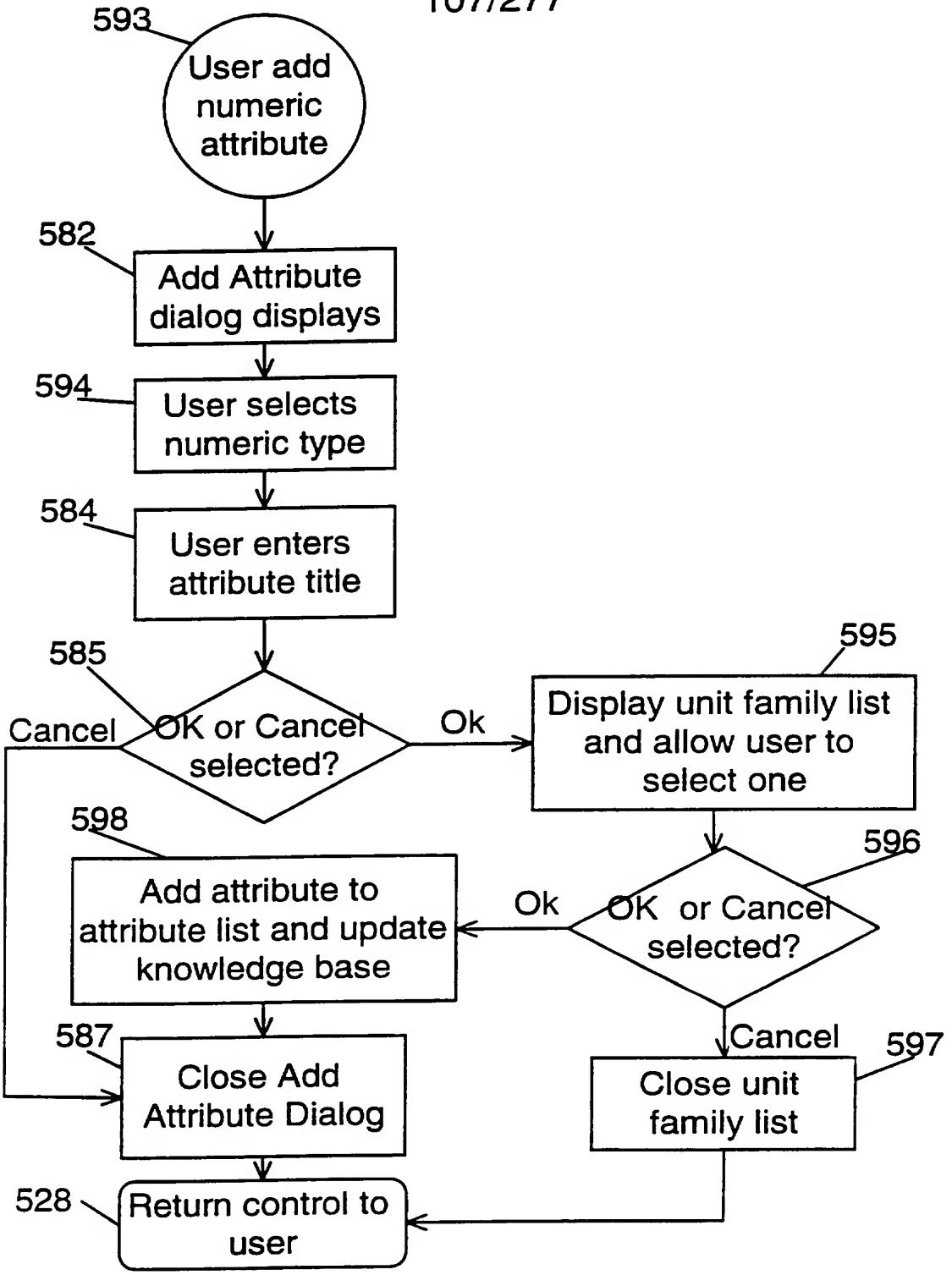

593
User add numeric attribute
582
Add Attribute dialog displays
594
User selects numeric type
584
User enters attribute title
585
OK or Cancel selected?
Cancel
Ok
595
Display unit family list and allow user to select one
596
OK or Cancel selected?
Ok
598
Add attribute to attribute list and update knowledge base
Cancel
597
Close unit family list
587
Close Add Attribute Dialog
528
Return control to user

FIG. 106

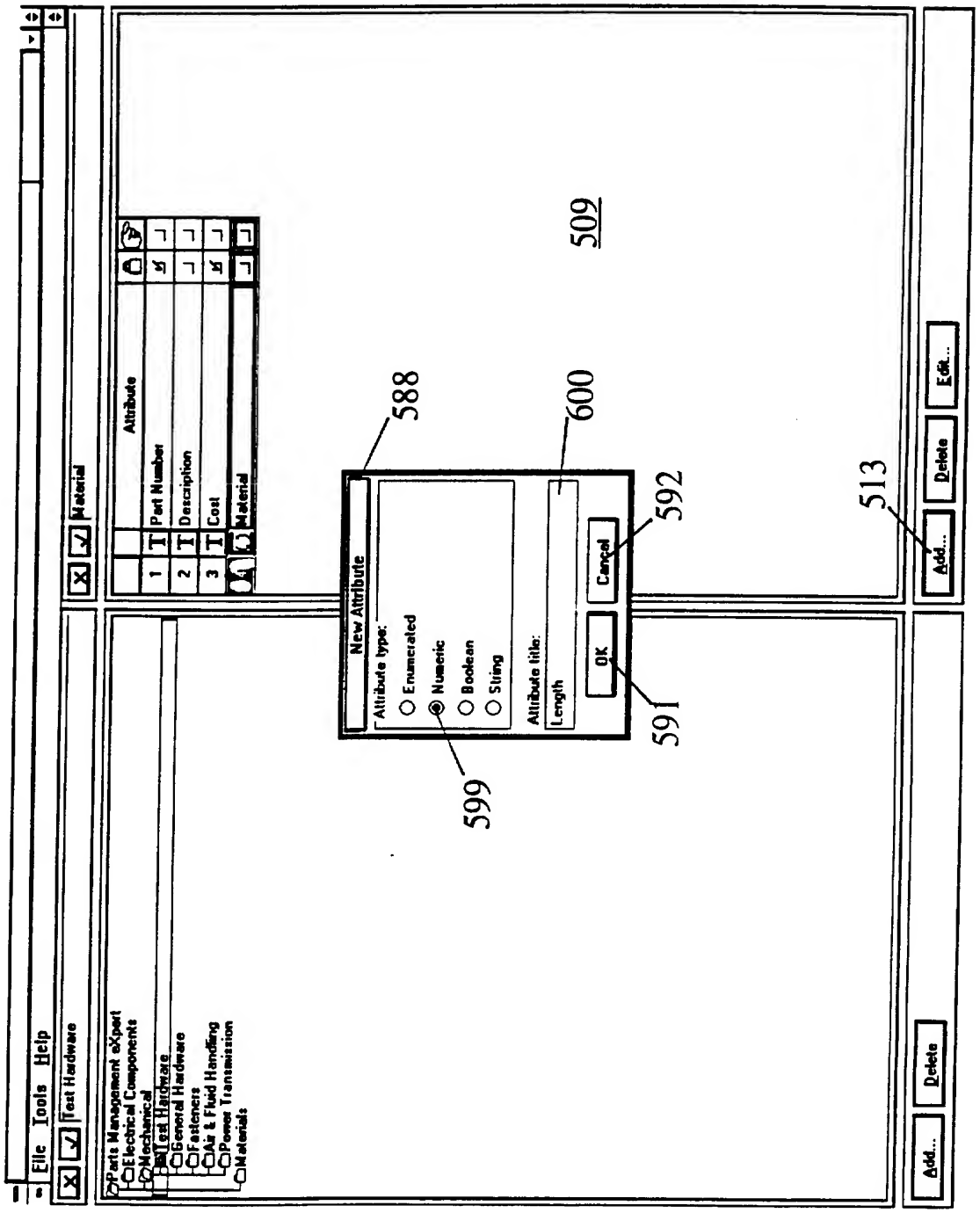

File Tools Help
Test Hardware
Material
Parts Management eXpert
Electrical Components
Mechanical
Test Hardware
General Hardware
Fasteners
Air & Fluid Handling
Power Transmission
Materials
Attribute
1 Part Number
2 Description
3 Cost
Material
509
588
600
592
513
599
591
New Attribute
Attribute type:
Enumerated
Numeric
Boolean
String
Attribute title:
Length
Cancel
OK
Add...
Delete
Edit...

FIG. 107

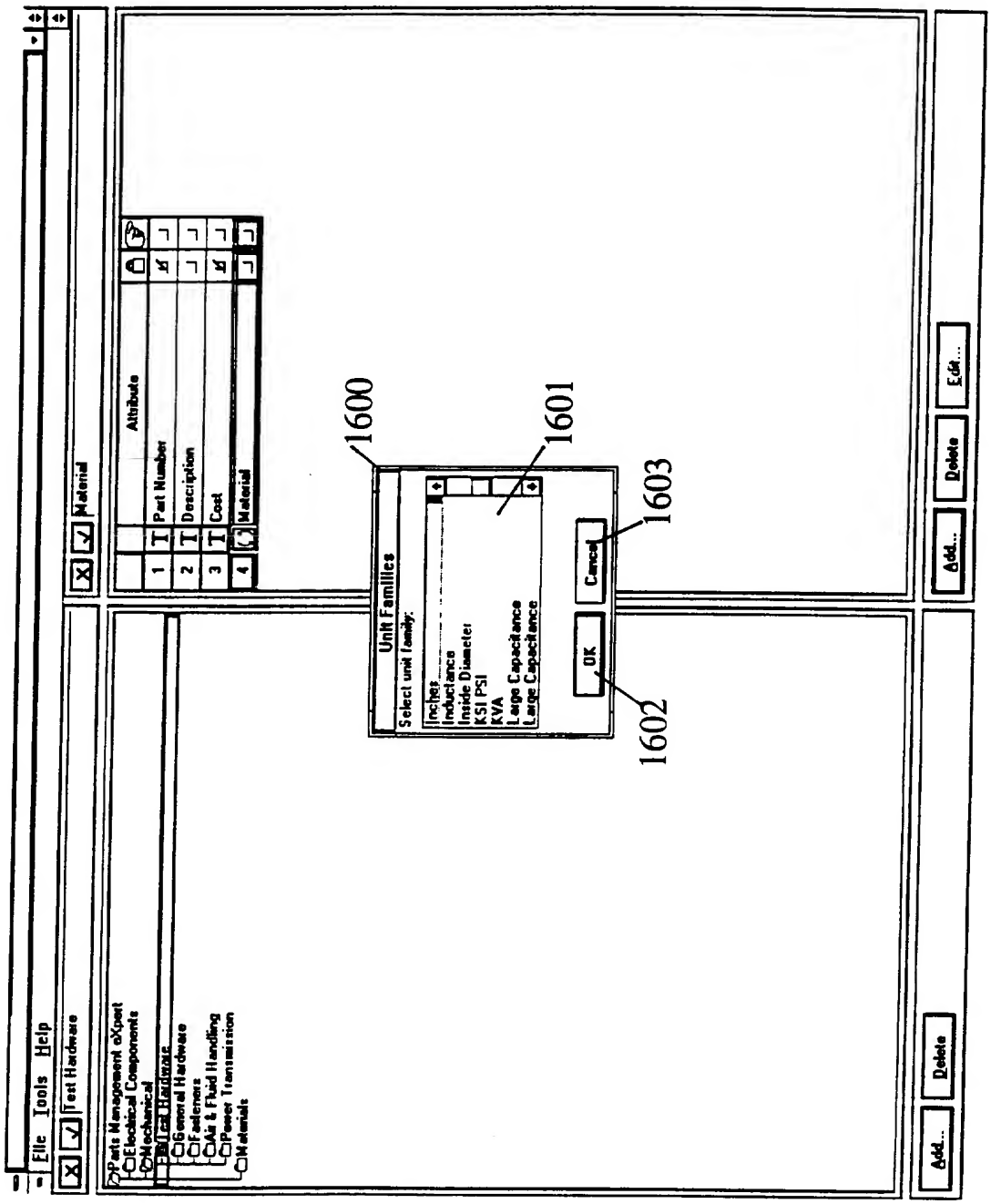

File Tools Help
Test Hardware
Material
Attribute
1 Part Number
2 Description
3 Cost
4 Material
Parts Management eXpert
Electrical Components
Mechanical
Test Hardware
General Hardware
Fasteners
Air & Fluid Handling
Power Transmission
Materials
Add...
Delete
Edit...
Unit Families
Select unit family:
Inches
Inductance
Inside Diameter
KSI PSI
KVA
Large Capacitance
Large Capacitance
OK
Cancel
1600
1601
1602
1603

FIG. 108

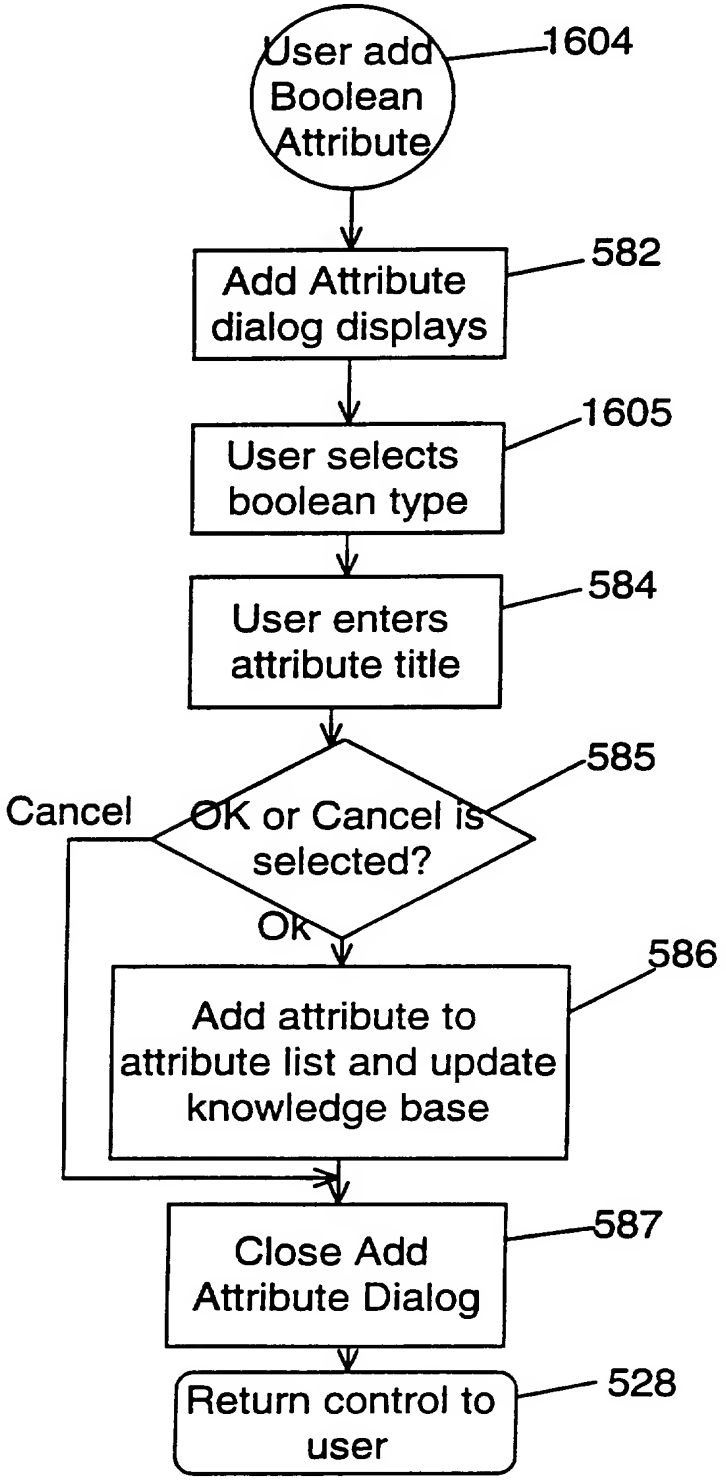

User add Boolean Attribute
1604
Add Attribute dialog displays
582
User selects boolean type
1605
User enters attribute title
584
OK or Cancel is selected?
585
Cancel
Ok
Add attribute to attribute list and update knowledge base
586
Close Add Attribute Dialog
587
Return control to user
528

FIG. 109

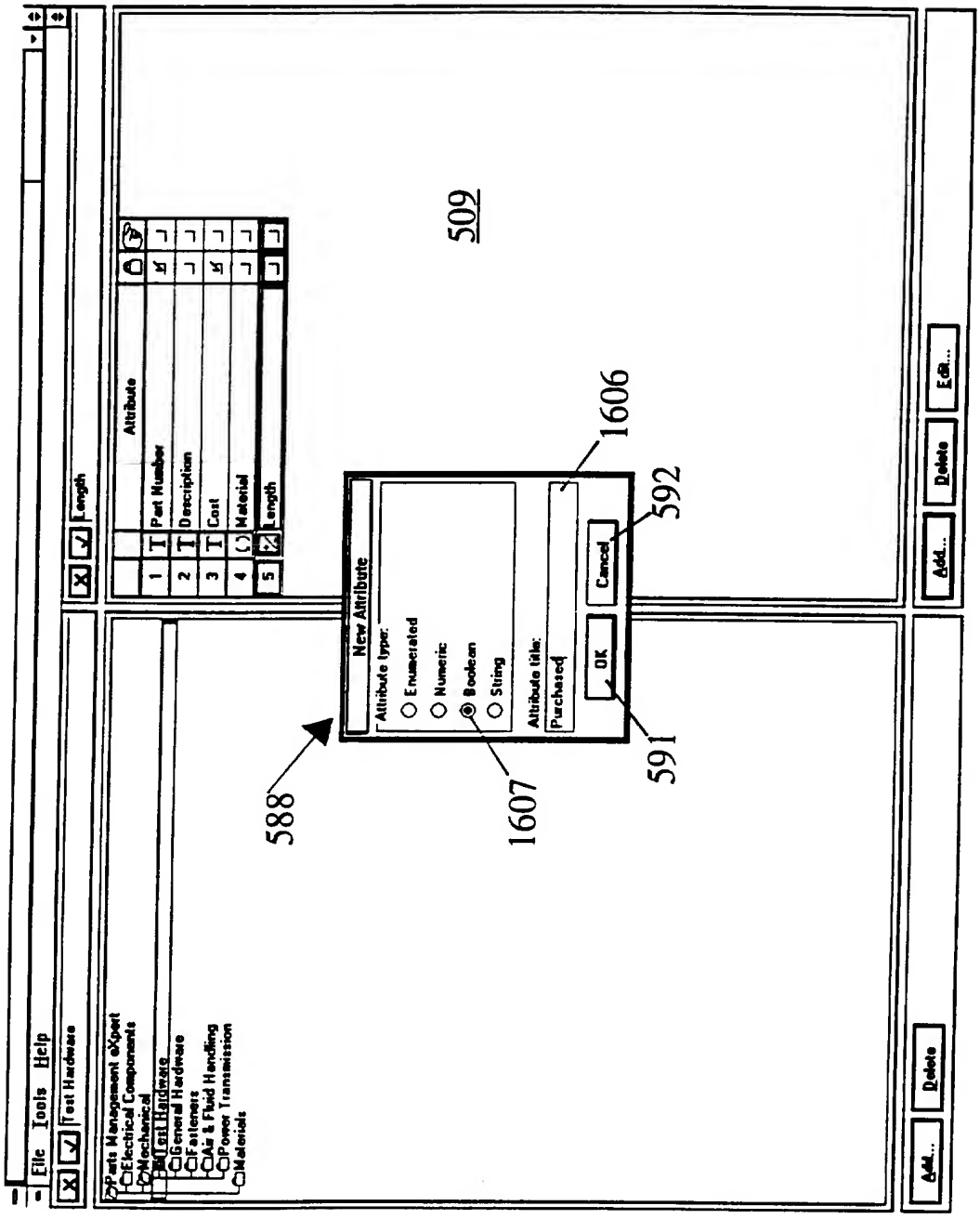

509
588
1607
1606
592
591
New Attribute
Attribute type:
Enumerated
Numeric
Boolean
String
Attribute title:
Purchased
OK
Cancel
Attribute
Part Number
Description
Cost
Material
Length
File Tools Help
Test Hardware
Length
Add...
Delete
Edit...

FIG. 110

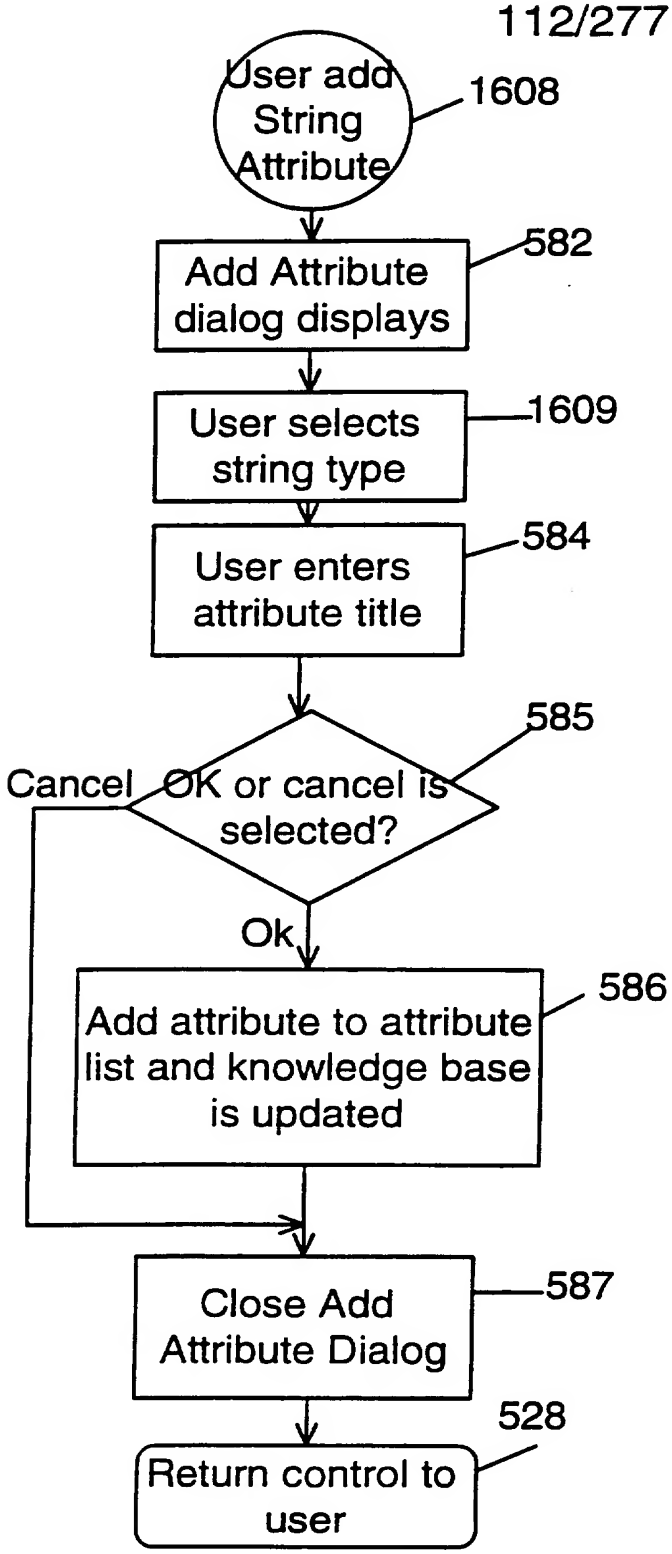

User add String Attribute
1608
Add Attribute dialog displays
582
User selects string type
1609
User enters attribute title
584
OK or cancel is selected?
585
Cancel
Ok
Add attribute to attribute list and knowledge base is updated
586
Close Add Attribute Dialog
587
Return control to user
528

FIG. 111

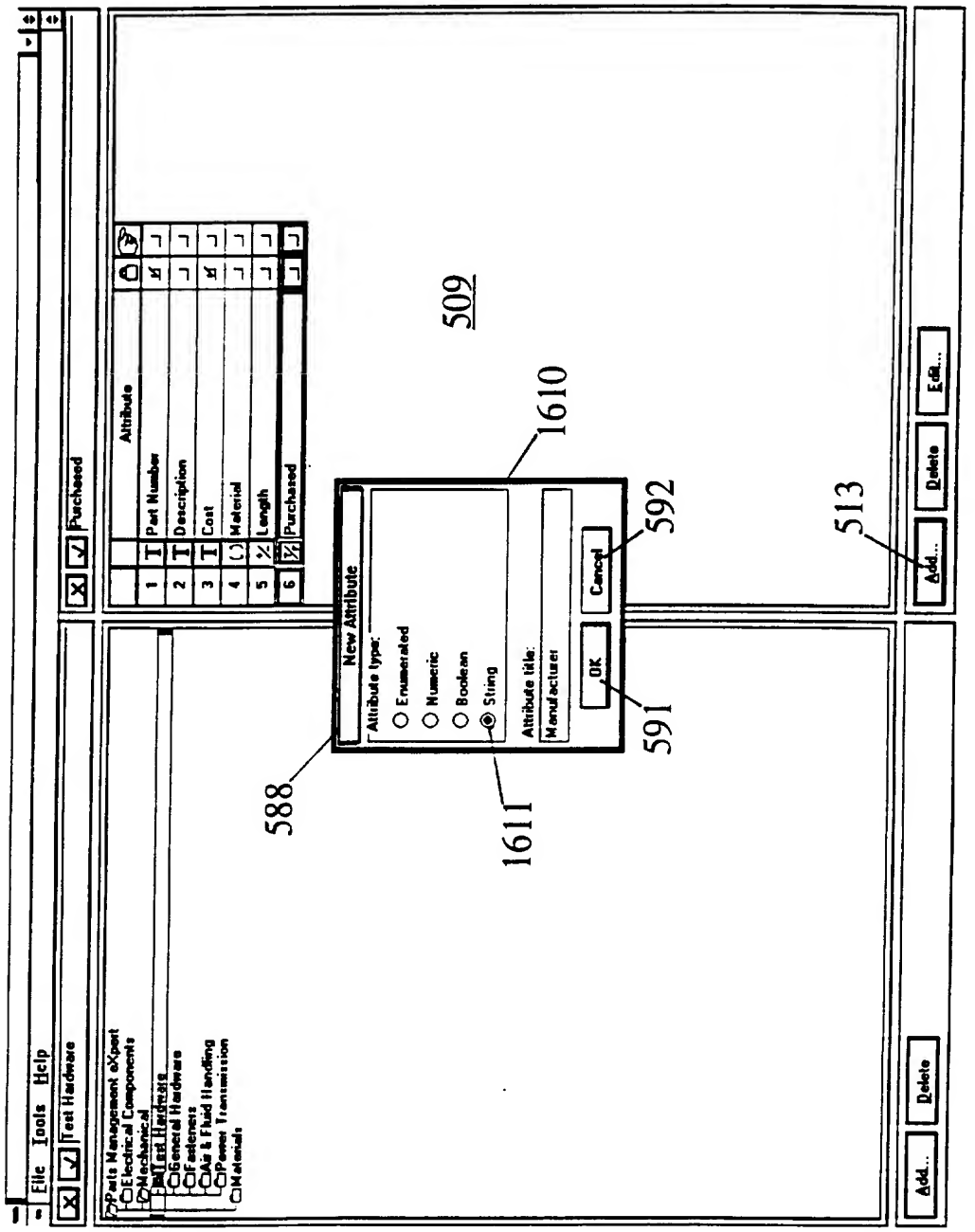

509
1610
592
513
588
1611
591
New Attribute
Attribute type:
Enumerated
Numeric
Boolean
String
Attribute title:
Manufacturer
OK
Cancel
Attribute
Part Number
Description
Cost
Material
Length
Purchased
Add...
Delete
Edit...
File Tools Help
Test Hardware

FIG. 112

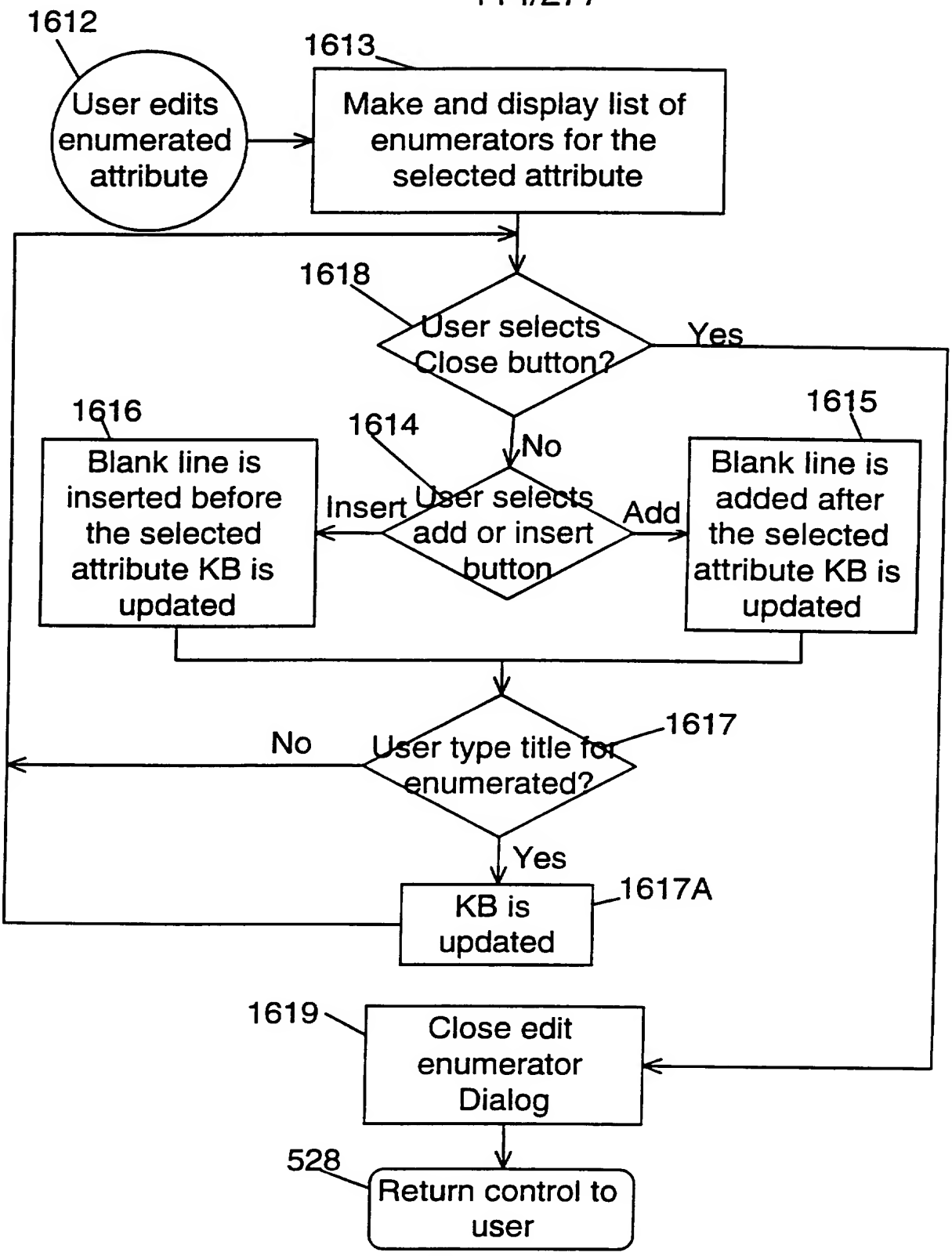

1612
User edits enumerated attribute
1613
Make and display list of enumerators for the selected attribute
1618
User selects Close button?
Yes
No
1614
User selects add or insert button
Insert
Add
1616
Blank line is inserted before the selected attribute KB is updated
1615
Blank line is added after the selected attribute KB is updated
1617
User type title for enumerated?
No
Yes
1617A
KB is updated
1619
Close edit enumerator Dialog
528
Return control to user

FIG. 113

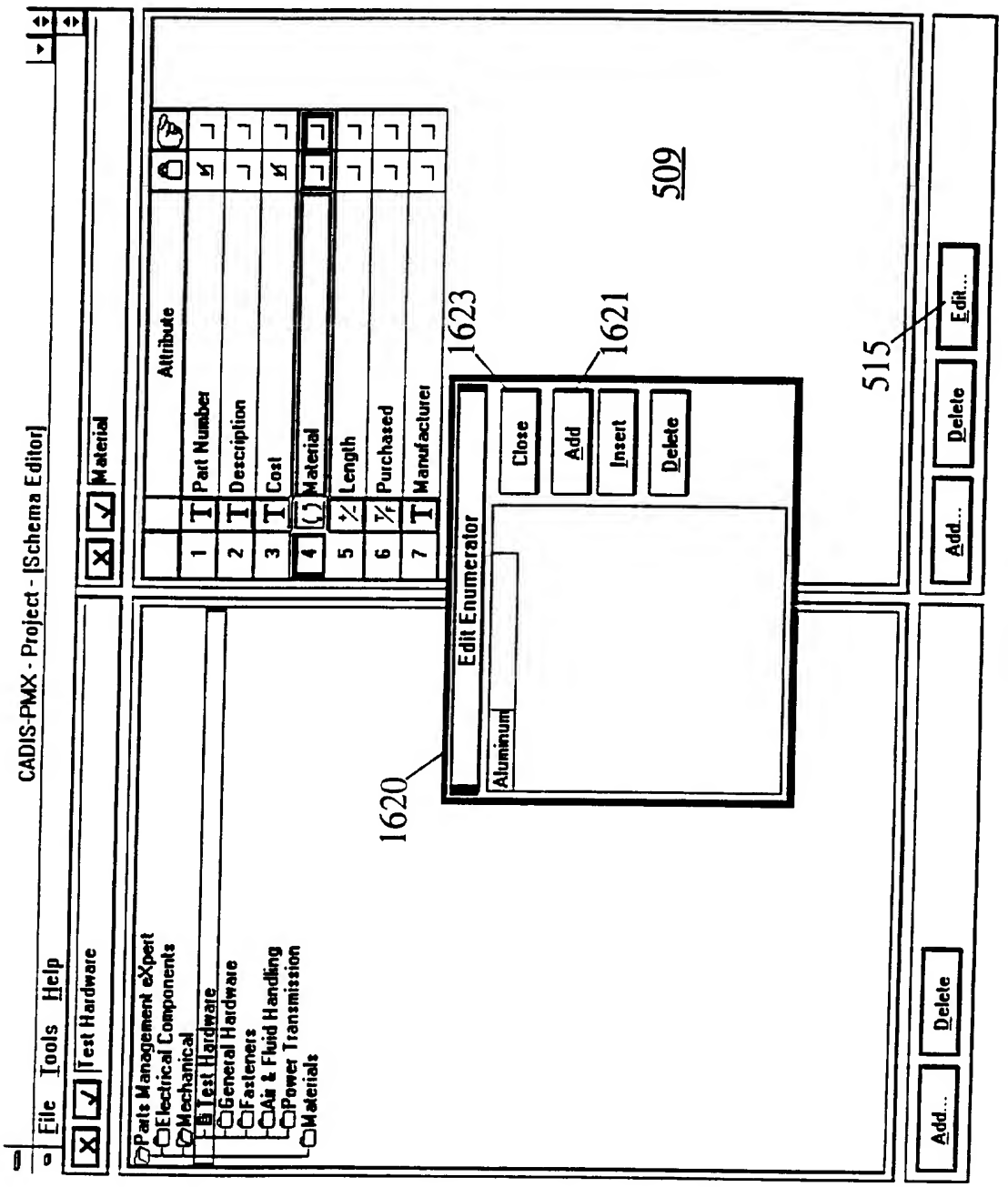

CADIS-PMX - Project - [Schema Editor]
File Tools Help
Test Hardware
Material
Parts Management eXpert
Electrical Components
Mechanical
Test Hardware
General Hardware
Fasteners
Air & Fluid Handling
Power Transmission
Materials
Attribute
Part Number
Description
Cost
Material
Length
Purchased
Manufacturer
Edit Enumerator
Aluminum
Close
Add
Insert
Delete
Add...
Delete
Add...
Delete
Edit...
509
515
1620
1621
1623

FIG. 114

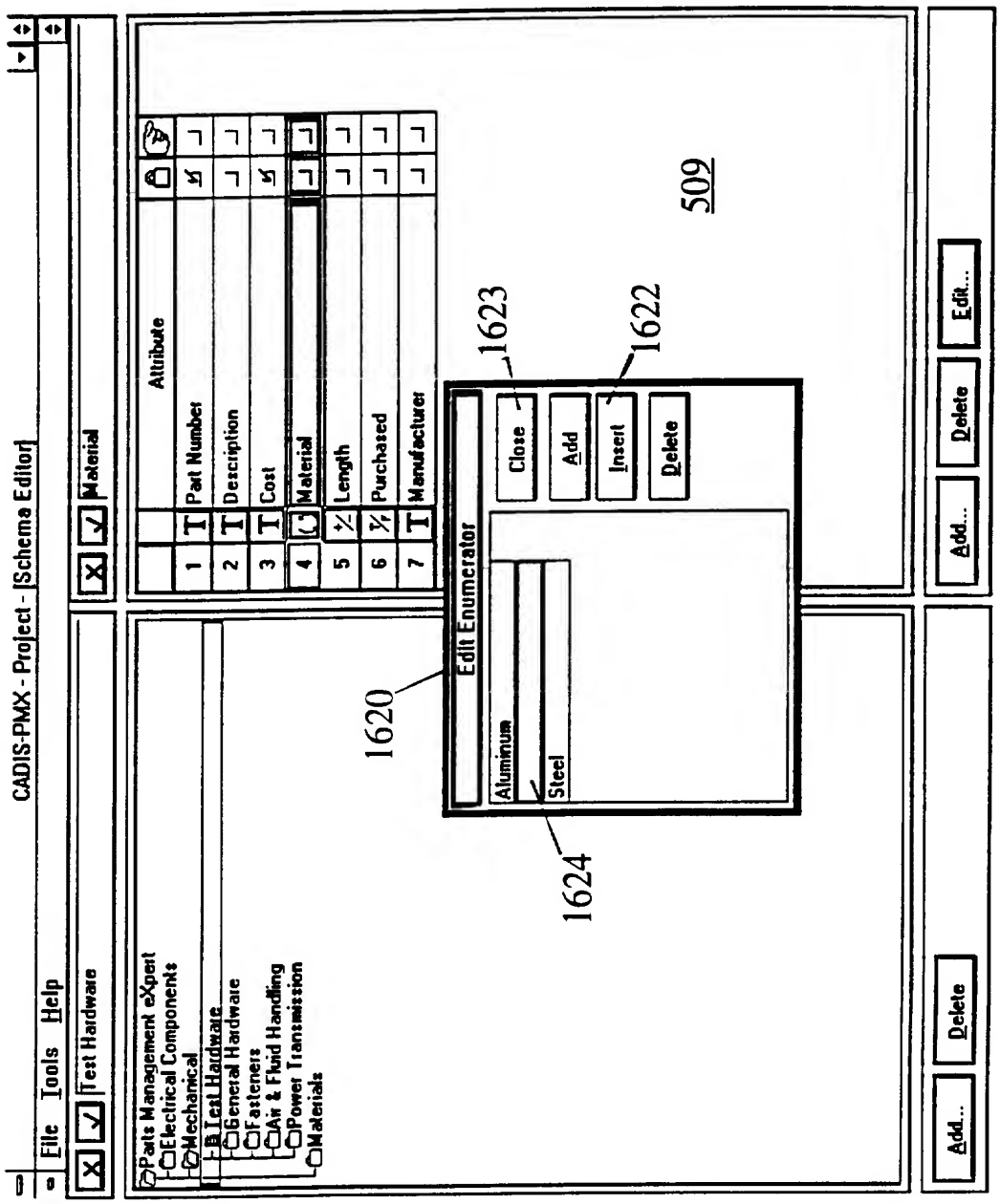

CADIS-PMX - Project - [Schema Editor]
File Tools Help
Test Hardware
Material
Parts Management eXpert
Electrical Components
Mechanical
Test Hardware
General Hardware
Fasteners
Air & Fluid Handling
Power Transmission
Materials
Attribute
1 Part Number
2 Description
3 Cost
4 Material
5 Length
6 Purchased
7 Manufacturer
Edit Enumerator
Aluminum
Steel
Close
Add
Insert
Delete
1620
1622
1623
1624
509
Add...
Delete
Add...
Delete
Edit...

FIG. 115

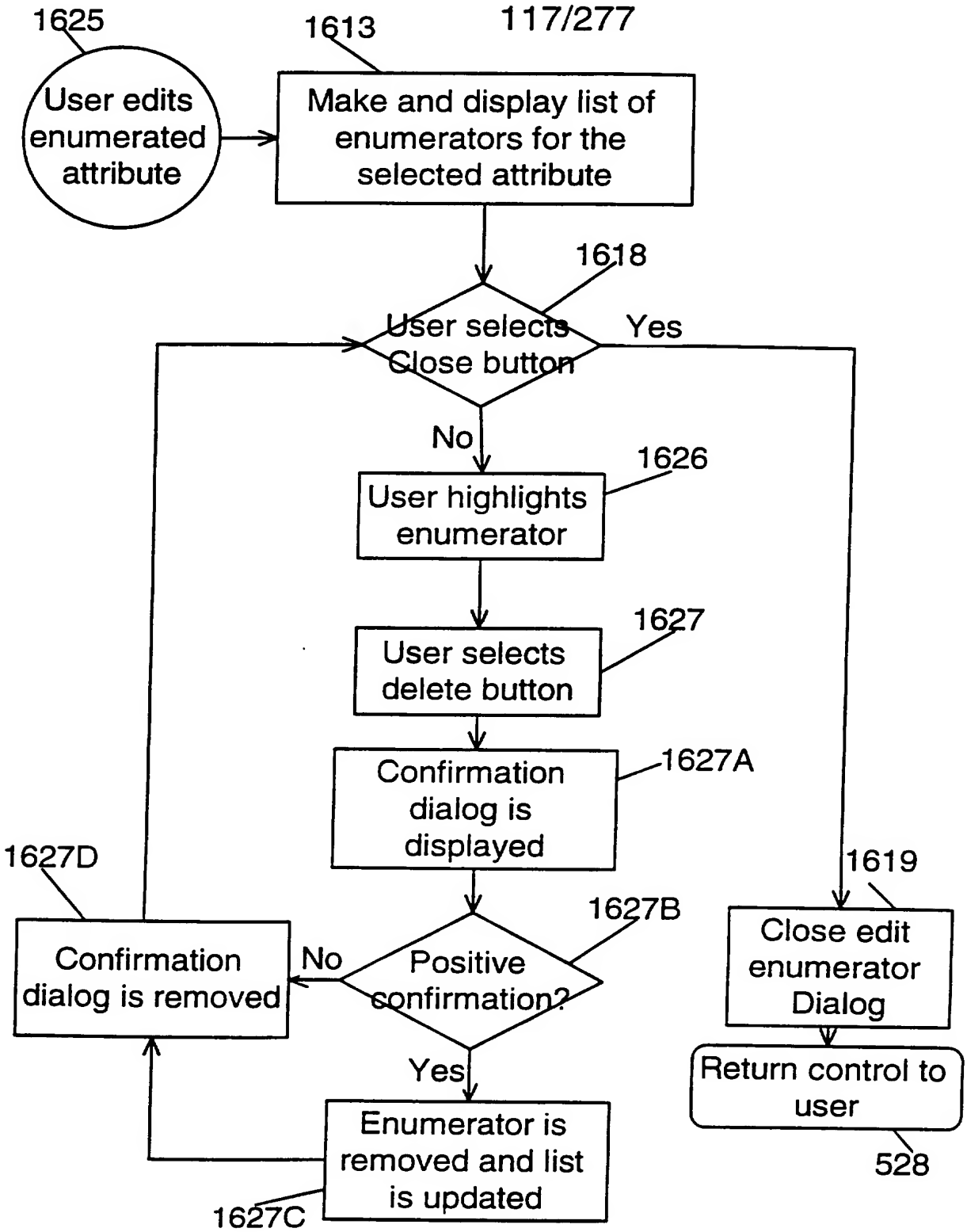

1625
User edits enumerated attribute
1613
Make and display list of enumerators for the selected attribute
1618
User selects Close button
Yes
No
1626
User highlights enumerator
1627
User selects delete button
1627A
Confirmation dialog is displayed
1627B
Positive confirmation?
No
1627D
Confirmation dialog is removed
Yes
Enumerator is removed and list is updated
1627C
1619
Close edit enumerator Dialog
Return control to user
528

FIG. 116

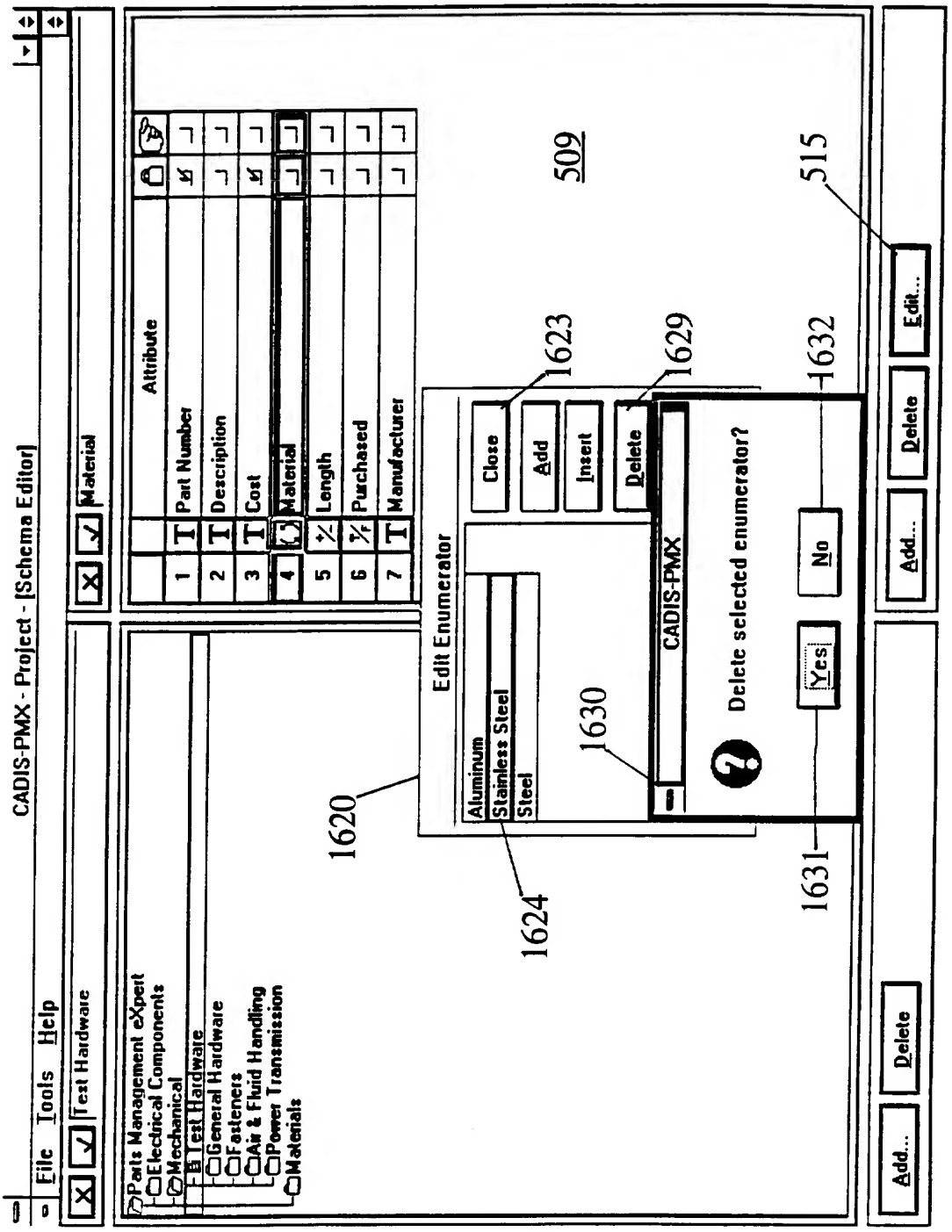

CADIS-PMX - Project - [Schema Editor]
File Tools Help
Test Hardware
Material
Parts Management eXpert
Electrical Components
Mechanical
Test Hardware
General Hardware
Fasteners
Air & Fluid Handling
Power Transmission
Materials
Attribute
1 Part Number
2 Description
3 Cost
4 Material
5 Length
6 Purchased
7 Manufacturer
509
Edit Enumerator
Aluminum
Stainless Steel
Steel
Close
Add
Insert
Delete
1623
1629
1632
515
1620
1630
1624
1631
CADIS-PMX
Delete selected enumerator?
Yes
No
Add...
Delete
Add...
Delete
Edit...

FIG. 117

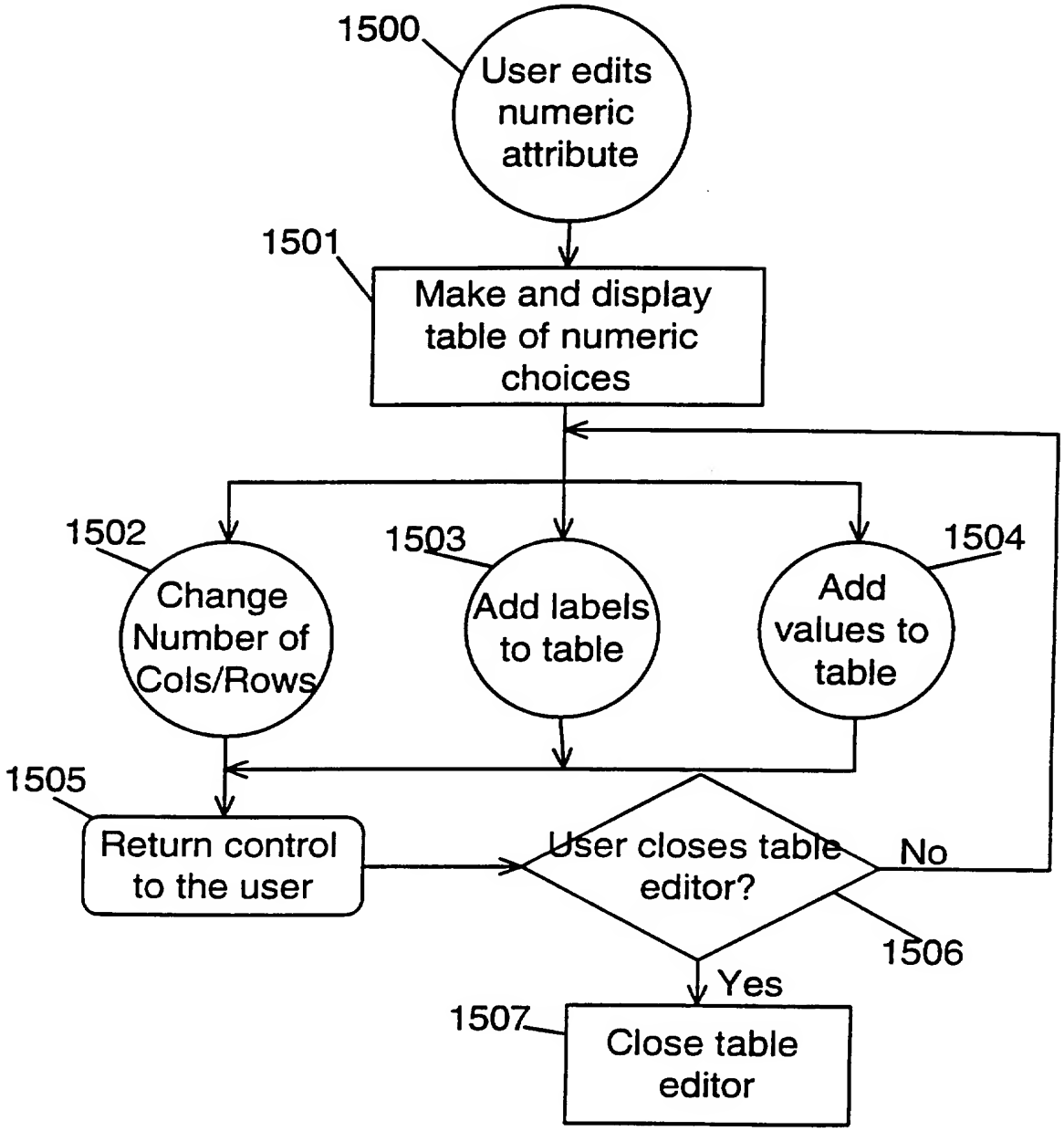

1500
User edits numeric attribute
1501
Make and display table of numeric choices
1502
Change Number of Cols/Rows
1503
Add labels to table
1504
Add values to table
1505
Return control to the user
User closes table editor?
No
1506
Yes
1507
Close table editor

FIG. 118

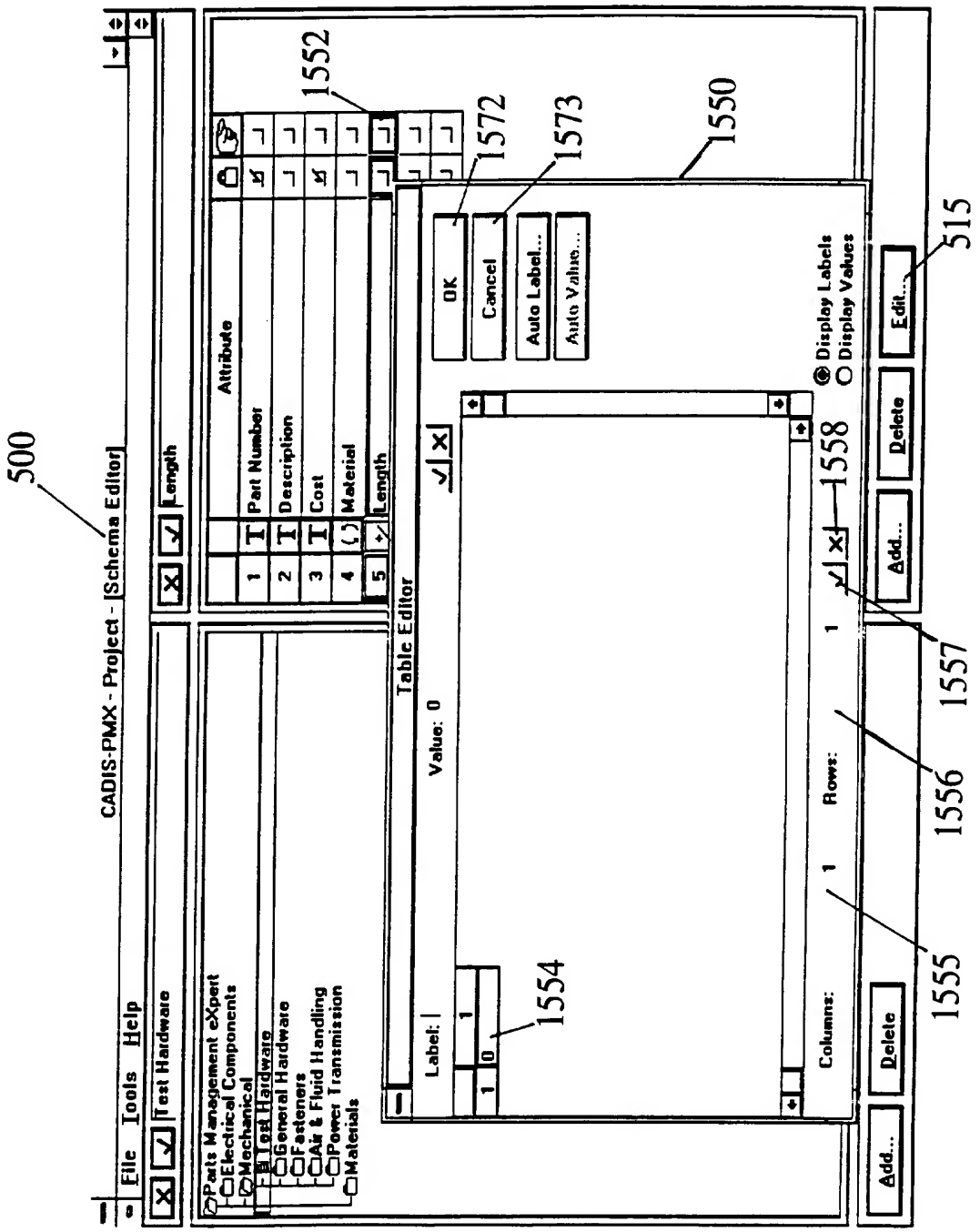

500
CADIS-PMX - Project - [Schema Editor]
File Tools Help
Test Hardware
Length
Parts Management eXpert
Electrical Components
Mechanical
Test Hardware
General Hardware
Fasteners
Air & Fluid Handling
Power Transmission
Materials
Attribute
1 Part Number
2 Description
3 Cost
4 Material
5 Length
1552
1572
1573
1550
OK
Cancel
Auto Label...
Auto Value...
Table Editor
Value: 0
Label:
1554
1558
Display Labels
Display Values
Edit...
515
Add...
Delete
Columns:
Rows:
1555
1556
1557

FIG. 119

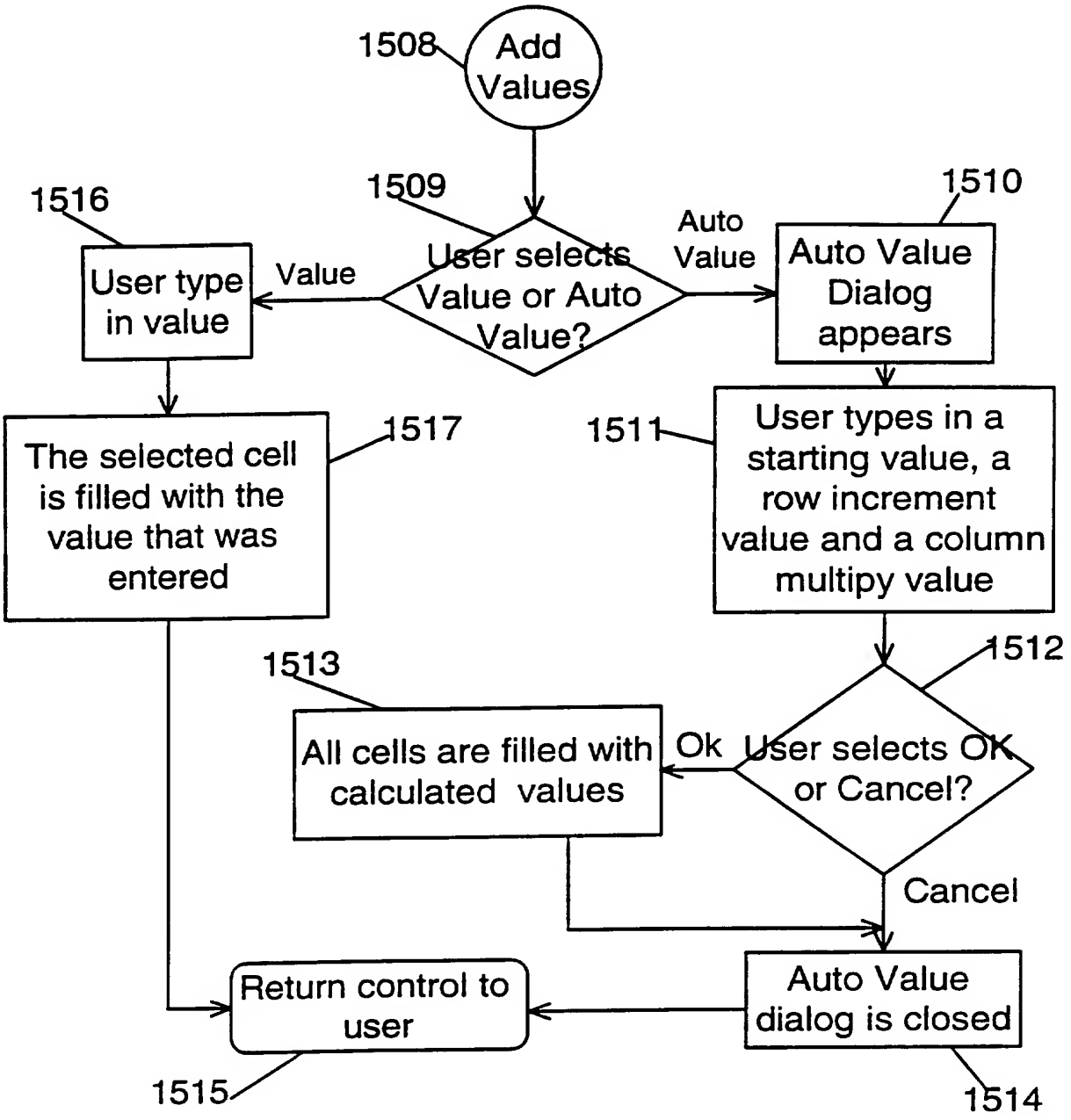

1508
Add Values
1509
User selects Value or Auto Value?
Value
Auto Value
1516
User type in value
1510
Auto Value Dialog appears
1517
The selected cell is filled with the value that was entered
1511
User types in a starting value, a row increment value and a column multipy value
1513
All cells are filled with calculated values
Ok
1512
User selects OK or Cancel?
Cancel
Return control to user
Auto Value dialog is closed
1515
1514

FIG. 120

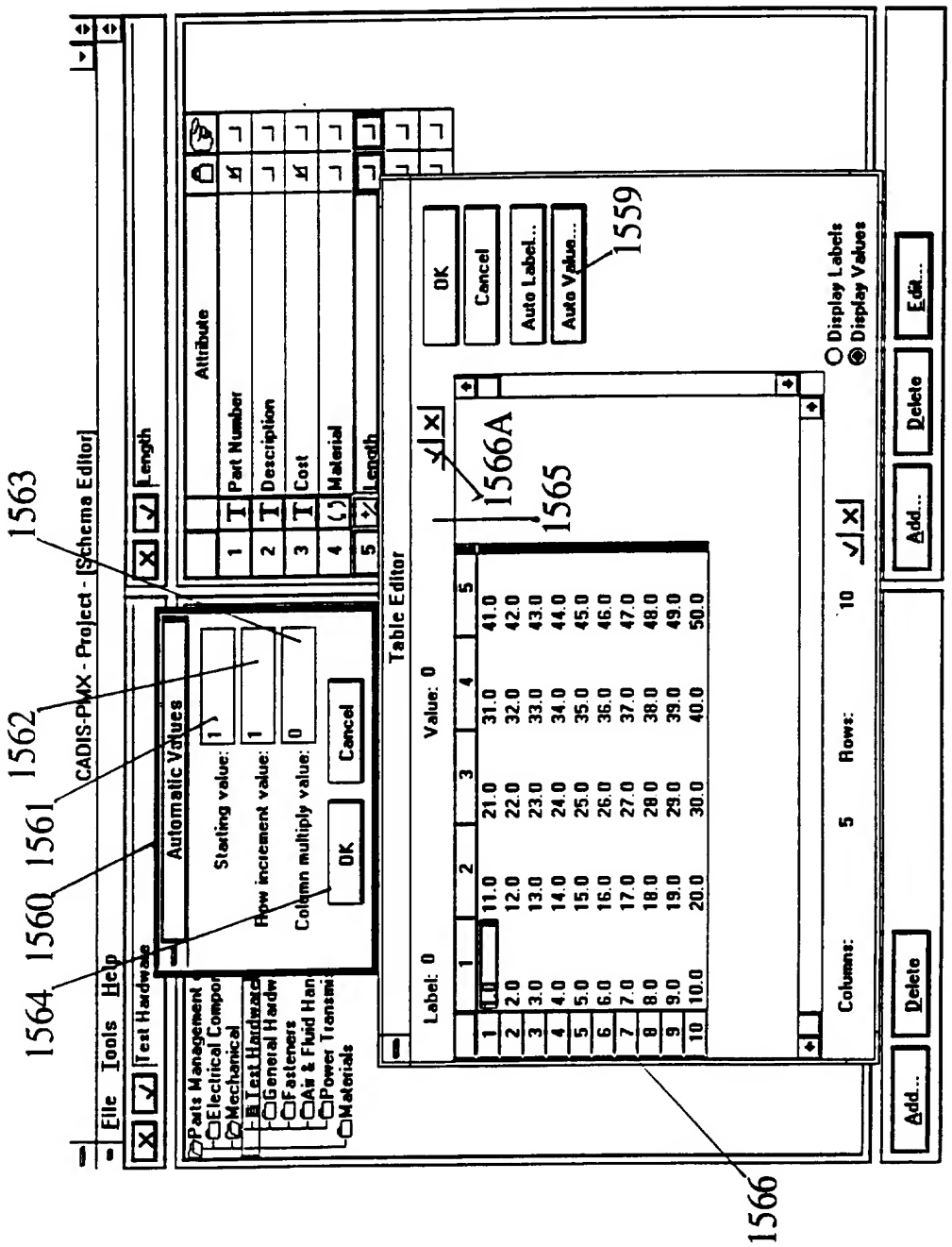

1563
1562
1561
1560
1564
1559
1566A
1565
1566
CADIS-PMX - Project - [Schema Editor]
File Tools Help
Test Hardware
Parts Management
Electrical Compor
Mechanical
Test Hardware
General Hardw
Fasteners
Air & Fluid Han
Power Transmi
Materials
Automatic Values
Starting value: 1
Row increment value: 1
Column multiply value: 0
OK
Cancel
Length
Attribute
1 Part Number
2 Description
3 Cost
4 Material
5 Length
Table Editor
Label: 0
Value: 0
OK
Cancel
Auto Label...
Auto Value...
Display Labels
Display Values
Columns: 5 Rows: 10
Add...
Delete
Edit...
Add...
Delete

FIG. 121

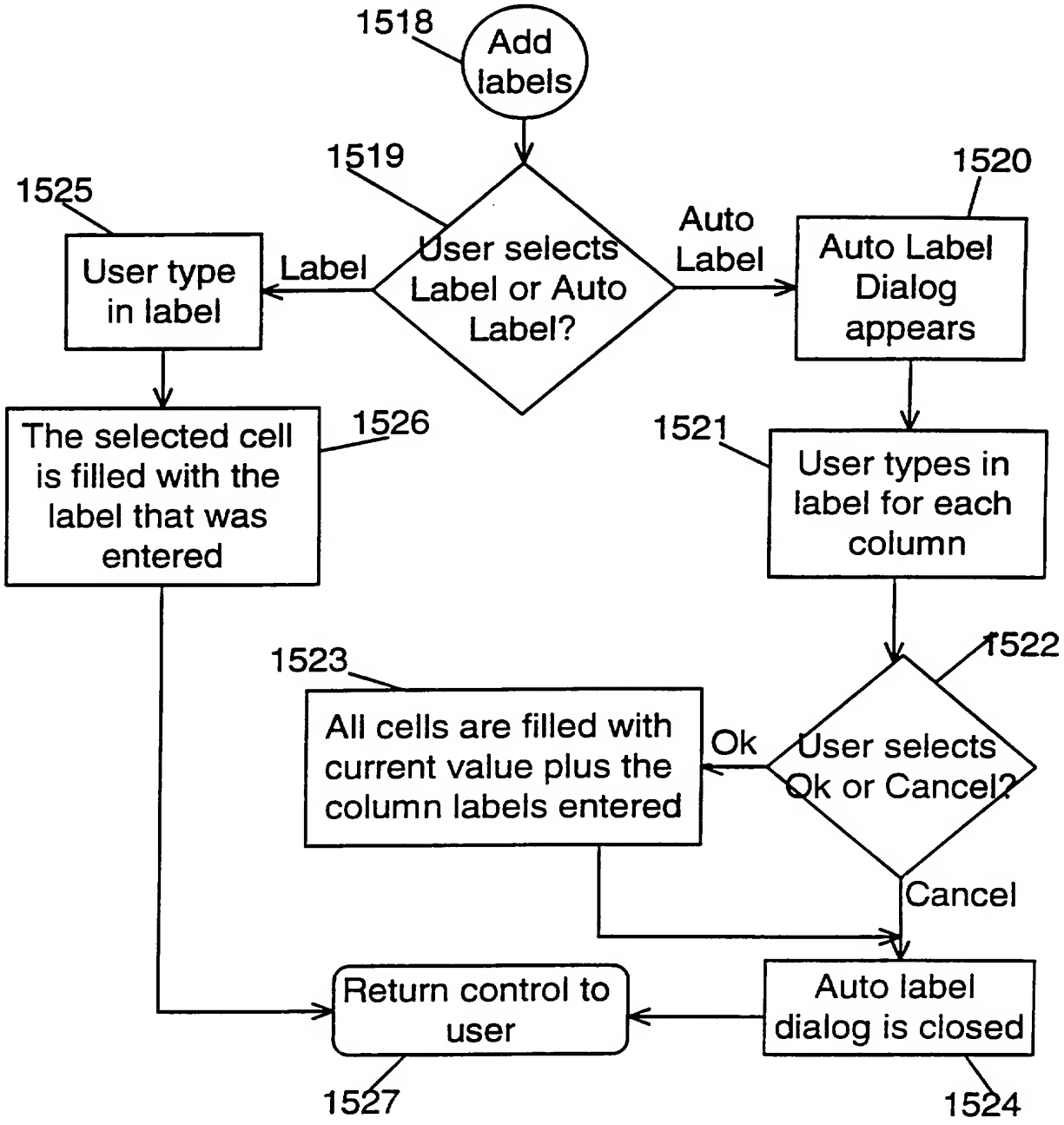

1518
Add labels
1519
User selects Label or Auto Label?
Label
Auto Label
1525
User type in label
1520
Auto Label Dialog appears
The selected cell is filled with the label that was entered
1526
1521
User types in label for each column
1523
All cells are filled with current value plus the column labels entered
Ok
1522
User selects Ok or Cancel?
Cancel
Return control to user
Auto label dialog is closed
1527
1524

FIG. 122

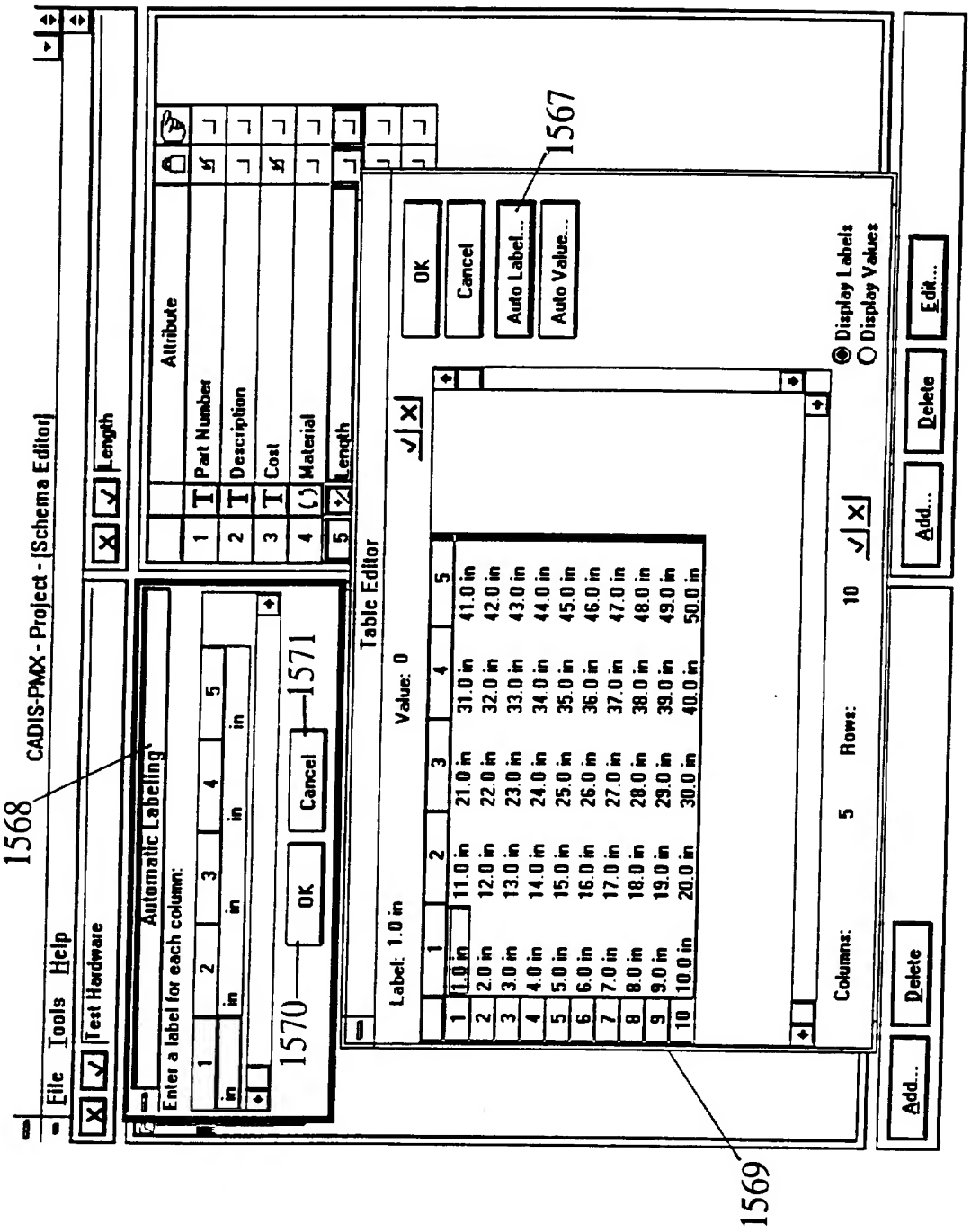

CADIS-PMX - Project - [Schema Editor]
File Tools Help
Test Hardware
Length
Attribute
Part Number
Description
Cost
Material
Length
Automatic Labelling
Enter a label for each column:
OK
Cancel
Table Editor
Label: 1.0 in
Value: 0
Auto Label...
Auto Value...
Display Labels
Display Values
Columns: 5
Rows: 10
Add...
Delete
Edit...
1567
1568
1569
1570
1571

FIG. 123

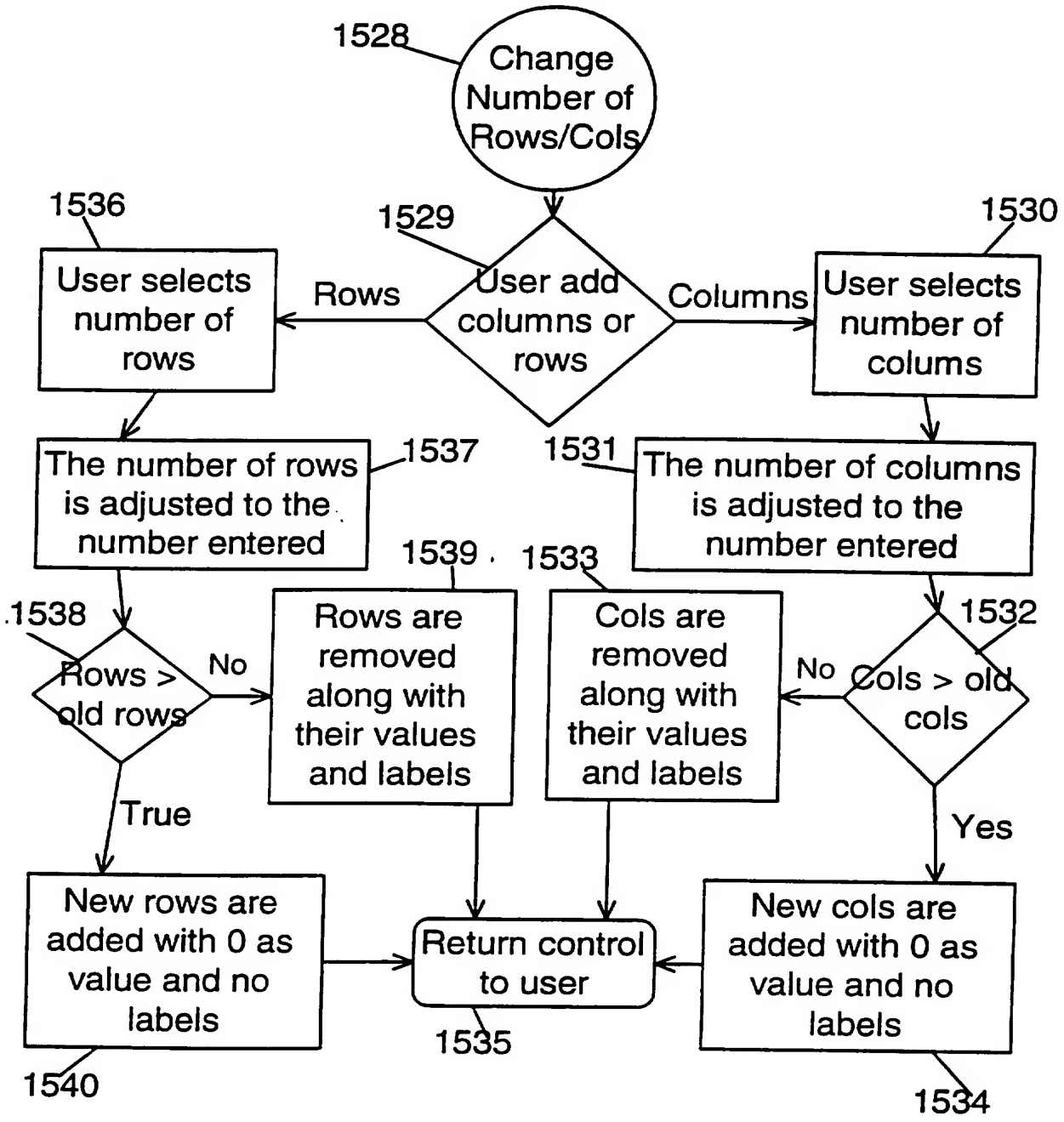

1528
Change Number of Rows/Cols
1529
User add columns or rows
Rows
Columns
1536
User selects number of rows
1530
User selects number of colums
1537
The number of rows is adjusted to the number entered
1531
The number of columns is adjusted to the number entered
1539
1533
1538
1532
Rows > old rows
No
Rows are removed along with their values and labels
Cols are removed along with their values and labels
No
Cols > old cols
True
Yes
New rows are added with 0 as value and no labels
Return control to user
New cols are added with 0 as value and no labels
1535
1540
1534

FIG. 124

import [-u user] [-p password] -d kdbname [-P] [-U] [-X] [-M] [-v] [-r] import_file	
-u	use user name 'username' when doing login, if none given login id is used
-p	use specified password with login,
-d	the logical database name to connect to
-P	turn on progress statistics
-U	don't import a parameter unless the instance is unique with respect to the search keys
-X	don't import a parameter unless there are no matching instances. Must be used in conjunction with -M.
-M	if a match is not found, make a new instance
-v	turn on verbose mode
-r	remove the matched instances from the knowledge base
import_file	name of the file containing the import info.

FIG. 125

simp [-u user] [-p password] -d kdbname -o outfile
[-f] [-P] [-U] [-v] [-n] import_file import_map

-u	use user name 'user name' when doing login, if none given login id is used
-p	use specified password with login
-d	the logical database name to connect to
-f	the field to match on.
-P	turn on progress statistics
-v	turn on verbose mode
-o	output file for instances not imported
-n	no mapfile - use defining class in import file as import class
import_file	name of the file containing the import info.
import_map	name of the file containing the map information

FIG. 126

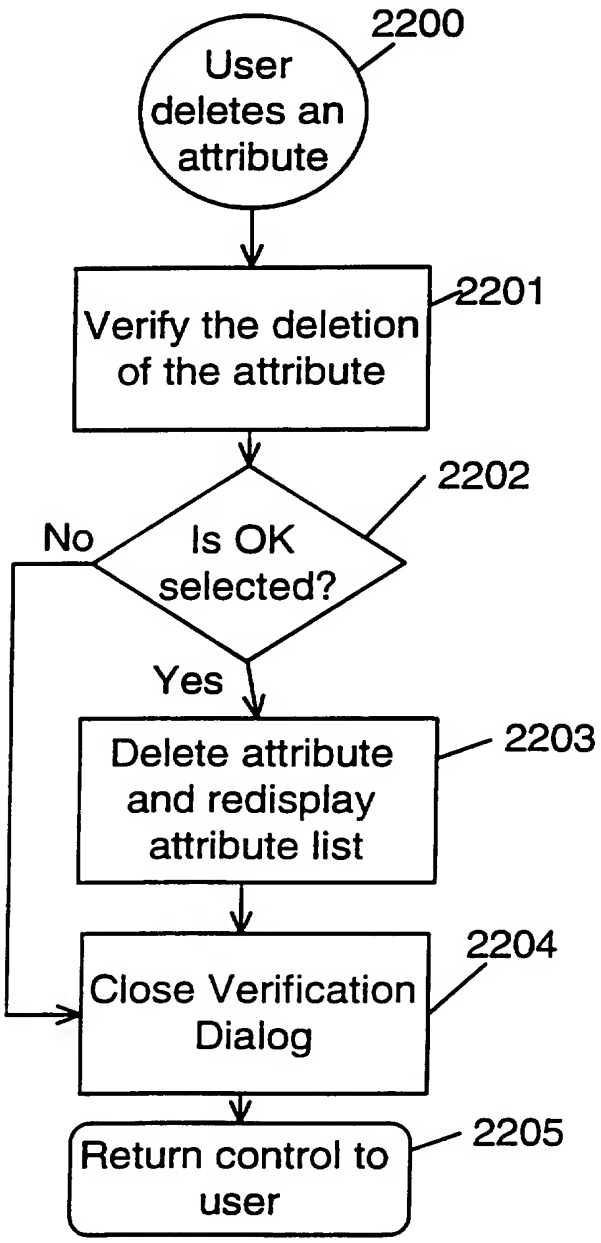

2200
User deletes an attribute
2201
Verify the deletion of the attribute
2202
Is OK selected?
No
Yes
2203
Delete attribute and redisplay attribute list
2204
Close Verification Dialog
2205
Return control to user

FIG. 127

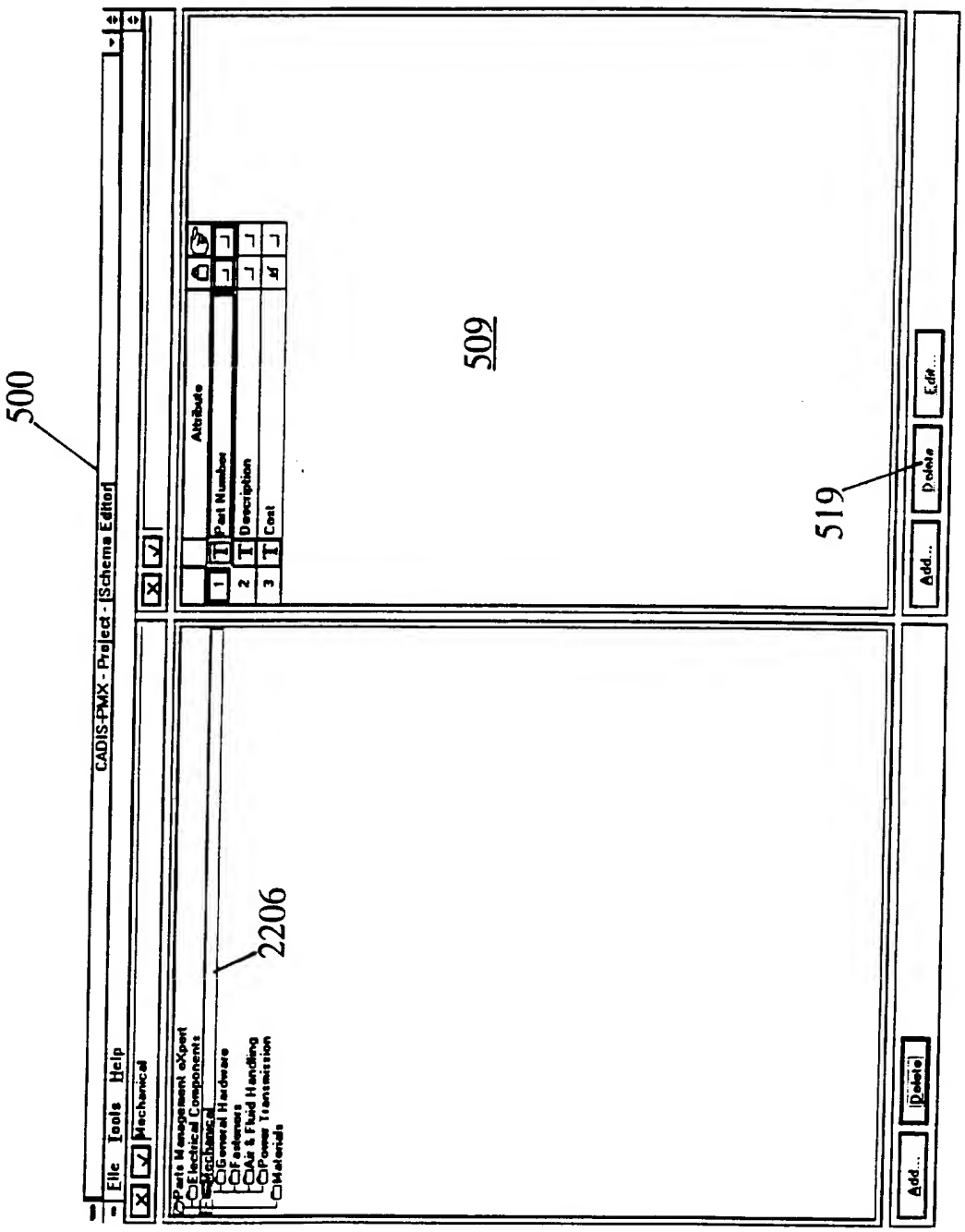

500
509
519
2206
CADIS-PMX - Project - [Schema Editor]
File Tools Help
Mechanical
Parts Management eXpert
Electrical Components
Mechanical
General Hardware
Fasteners
Air & Fluid Handling
Power Transmission
Materials
Attribute
Part Number
Description
Cost
Add...
Delete
Edit...

FIG. 128

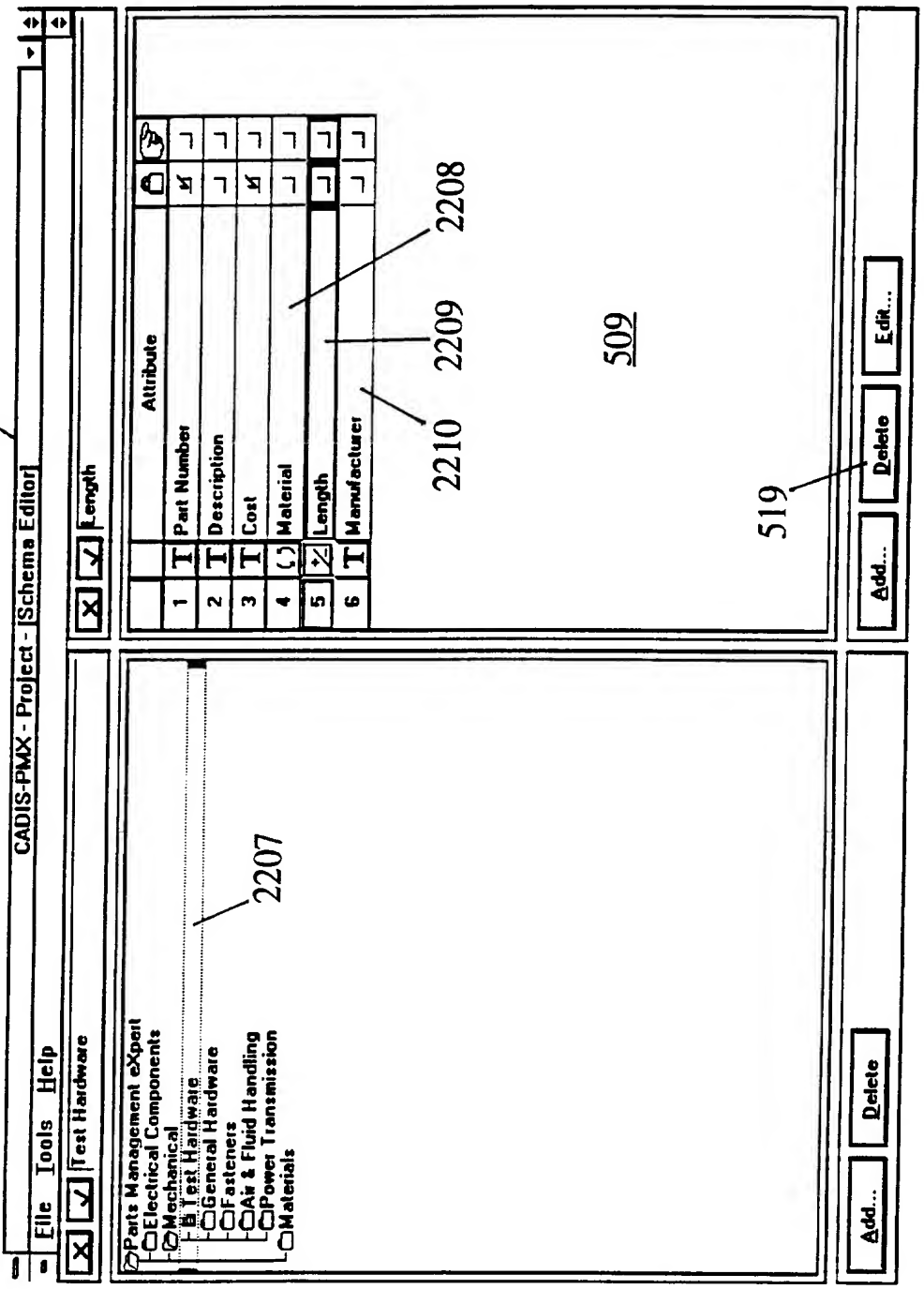

500
CADIS-PMX - Project - [Schema Editor]
File Tools Help
Test Hardware
Length
Attribute
1 Part Number
2 Description
3 Cost
4 Material
5 Length
6 Manufacturer
2208
2209
2210
509
519
2207
Parts Management eXpert
Electrical Components
Mechanical
Test Hardware
General Hardware
Fasteners
Air & Fluid Handling
Power Transmission
Materials
Add...
Delete
Add...
Delete
Edit...

FIG. 129

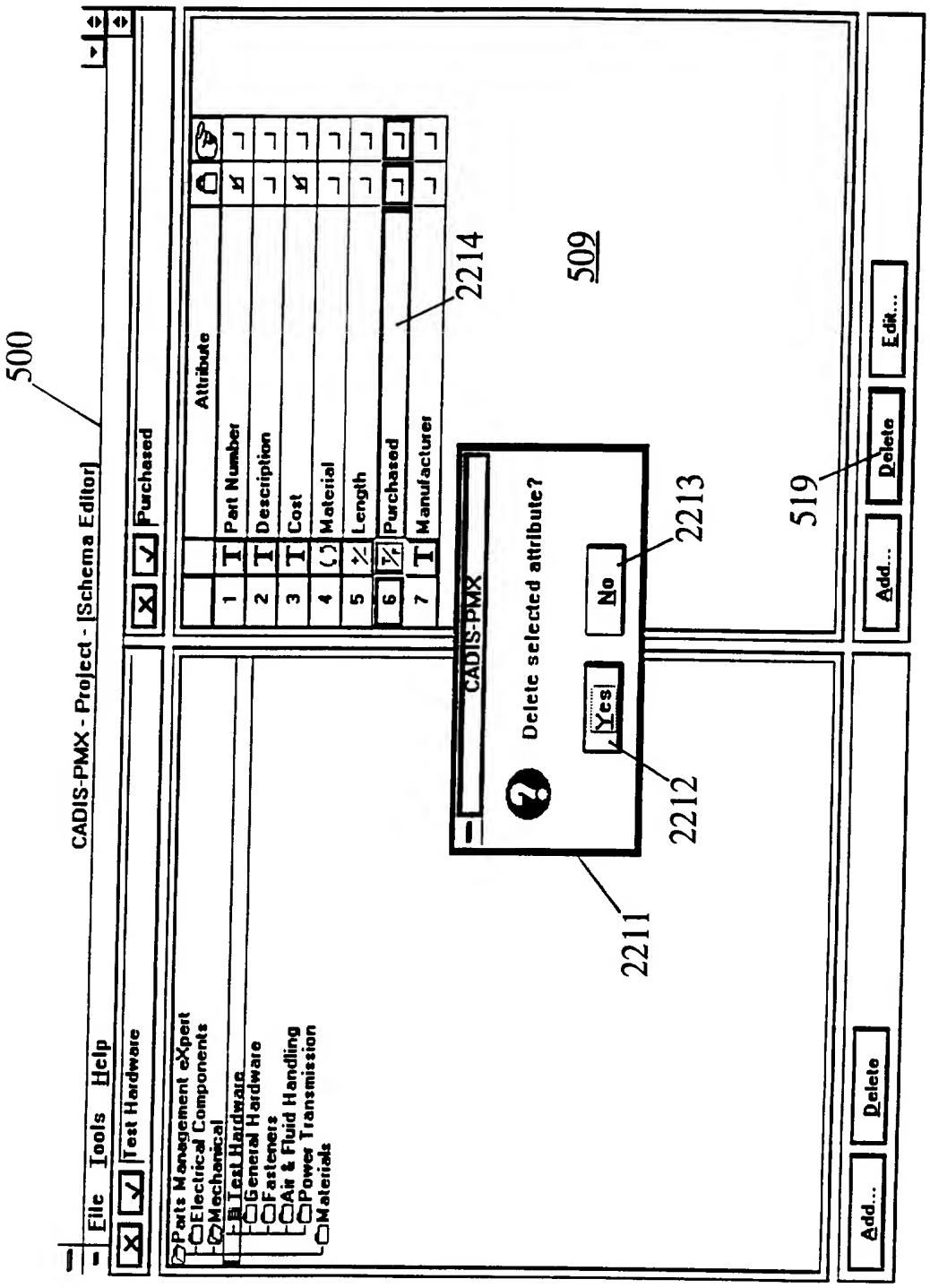

500
CADIS-PMX - Project - [Schema Editor]
File Tools Help
Test Hardware
Purchased
Parts Management eXpert
Electrical Components
Mechanical
Test Hardware
General Hardware
Fasteners
Air & Fluid Handling
Power Transmission
Materials
Attribute
1 Part Number
2 Description
3 Cost
4 Material
5 Length
6 Purchased
7 Manufacturer
2214
509
CADIS-PMX
Delete selected attribute?
Yes
No
2213
2212
2211
519
Add...
Delete
Edit...
Add...
Delete

FIG. 130

Match Component	Component	Matched?
Base Number	2901	
Prefix		Yes
Suffix	A	No
Manufacturer	Intel	Yes
# of Classes Found	1	Yes

FIG. 131

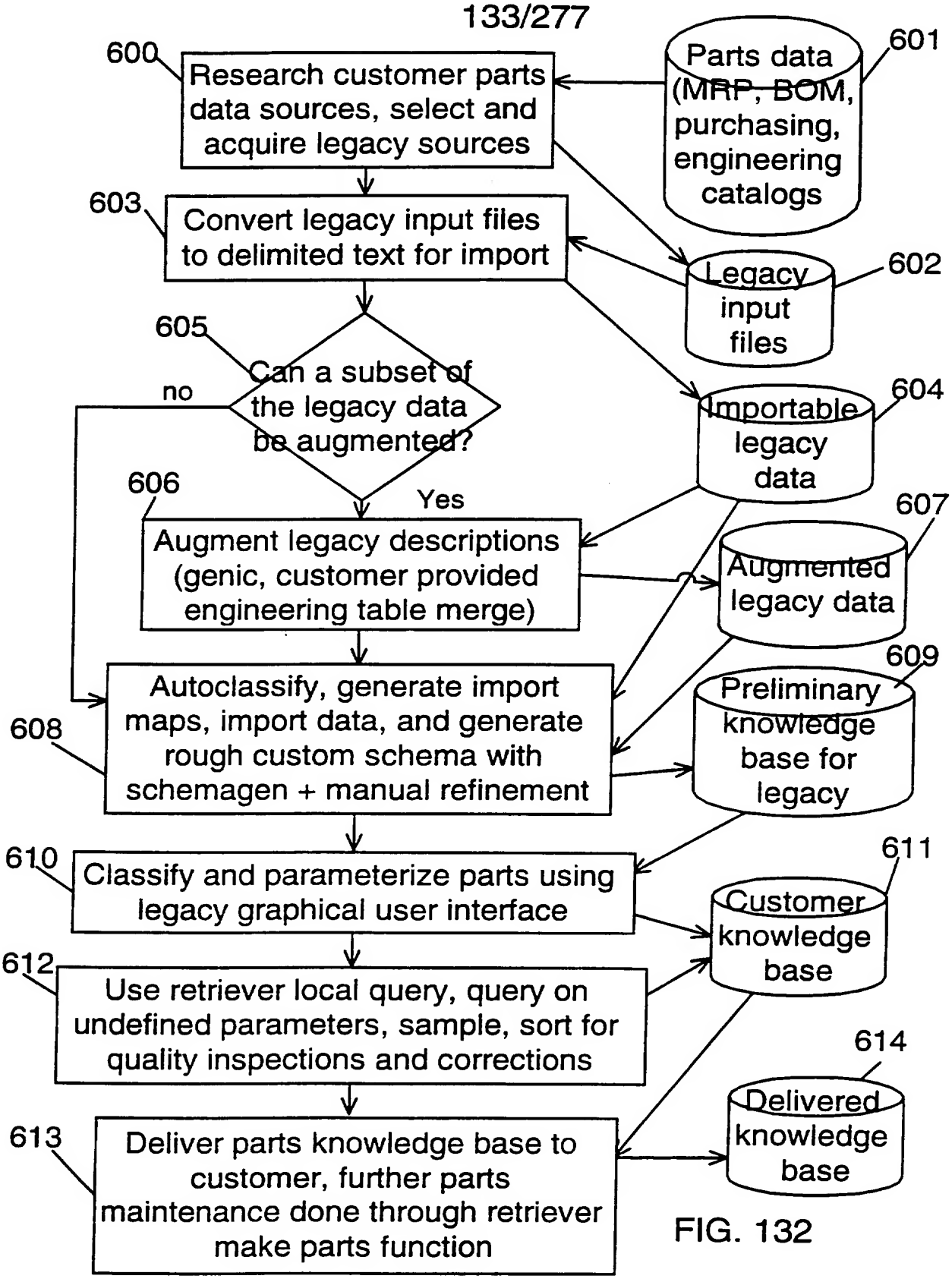

600
Research customer parts data sources, select and acquire legacy sources
601
Parts data (MRP, BOM, purchasing, engineering catalogs
603
Convert legacy input files to delimited text for import
602
Legacy input files
605
Can a subset of the legacy data be augmented?
no
Yes
604
Importable legacy data
606
Augment legacy descriptions (genic, customer provided engineering table merge)
607
Augmented legacy data
608
Autoclassify, generate import maps, import data, and generate rough custom schema with schemagen + manual refinement
609
Preliminary knowledge base for legacy
610
Classify and parameterize parts using legacy graphical user interface
611
Customer knowledge base
612
Use retriever local query, query on undefined parameters, sample, sort for quality inspections and corrections
614
Delivered knowledge base
613
Deliver parts knowledge base to customer, further parts maintenance done through retriever make parts function

FIG. 132

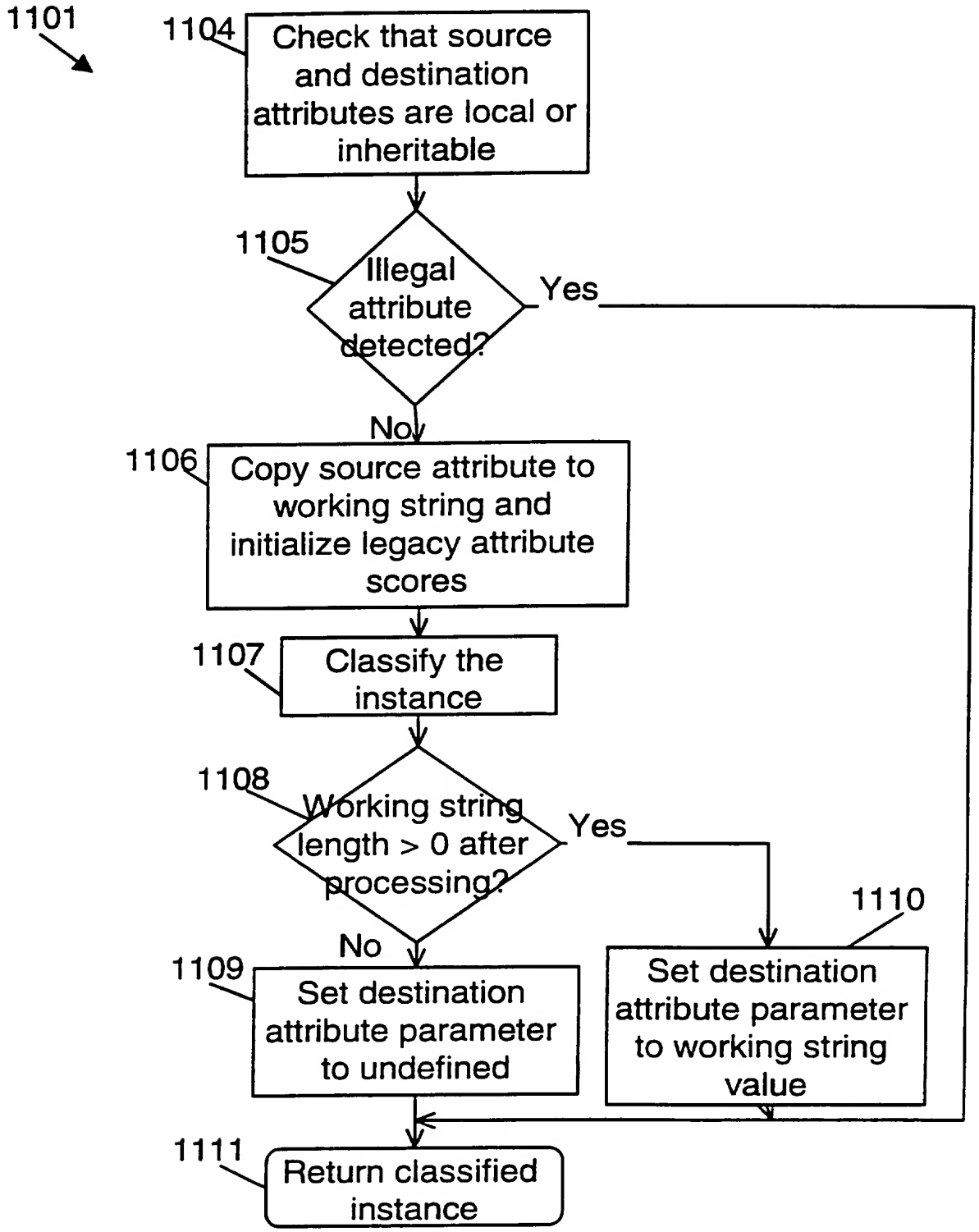

1101
1104
Check that source and destination attributes are local or inheritable
1105
Illegal attribute detected?
Yes
No
1106
Copy source attribute to working string and initialize legacy attribute scores
1107
Classify the instance
1108
Working string length > 0 after processing?
Yes
No
1109
Set destination attribute parameter to undefined
1110
Set destination attribute parameter to working string value
1111
Return classified instance

FIG.133

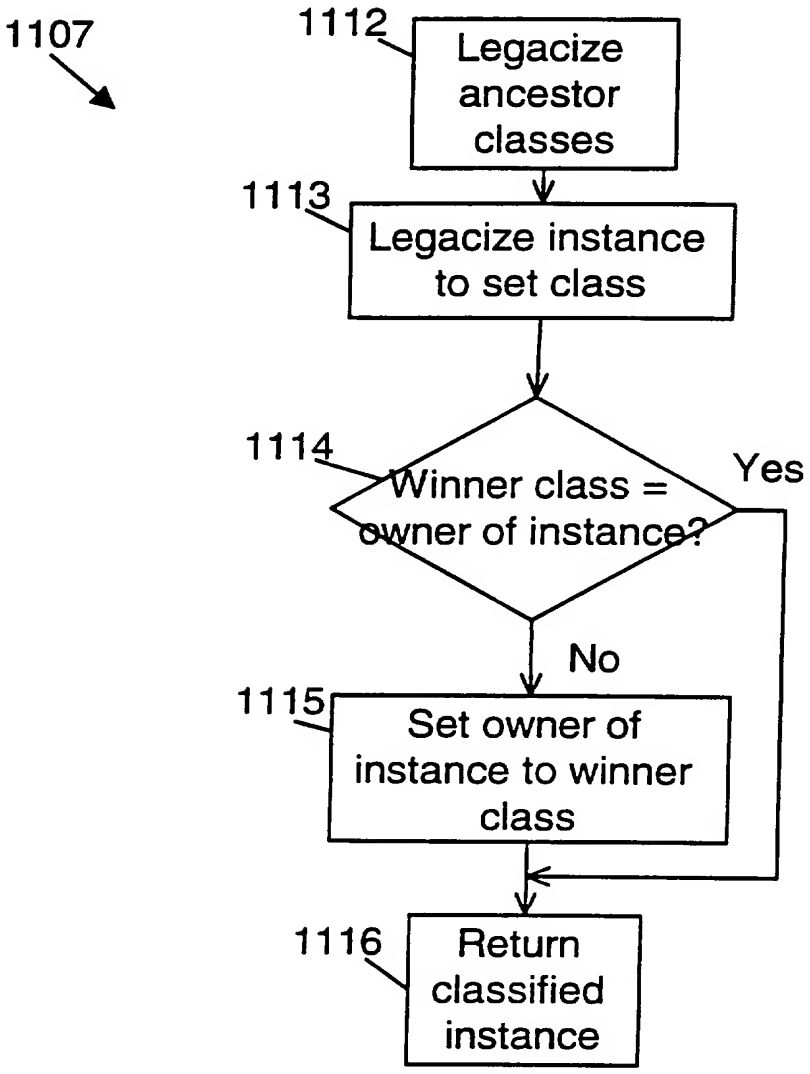

1107
1112
Legacize ancestor classes
1113
Legacize instance to set class
1114
Winner class = owner of instance?
Yes
No
1115
Set owner of instance to winner class
1116
Return classified instance

FIG. 134

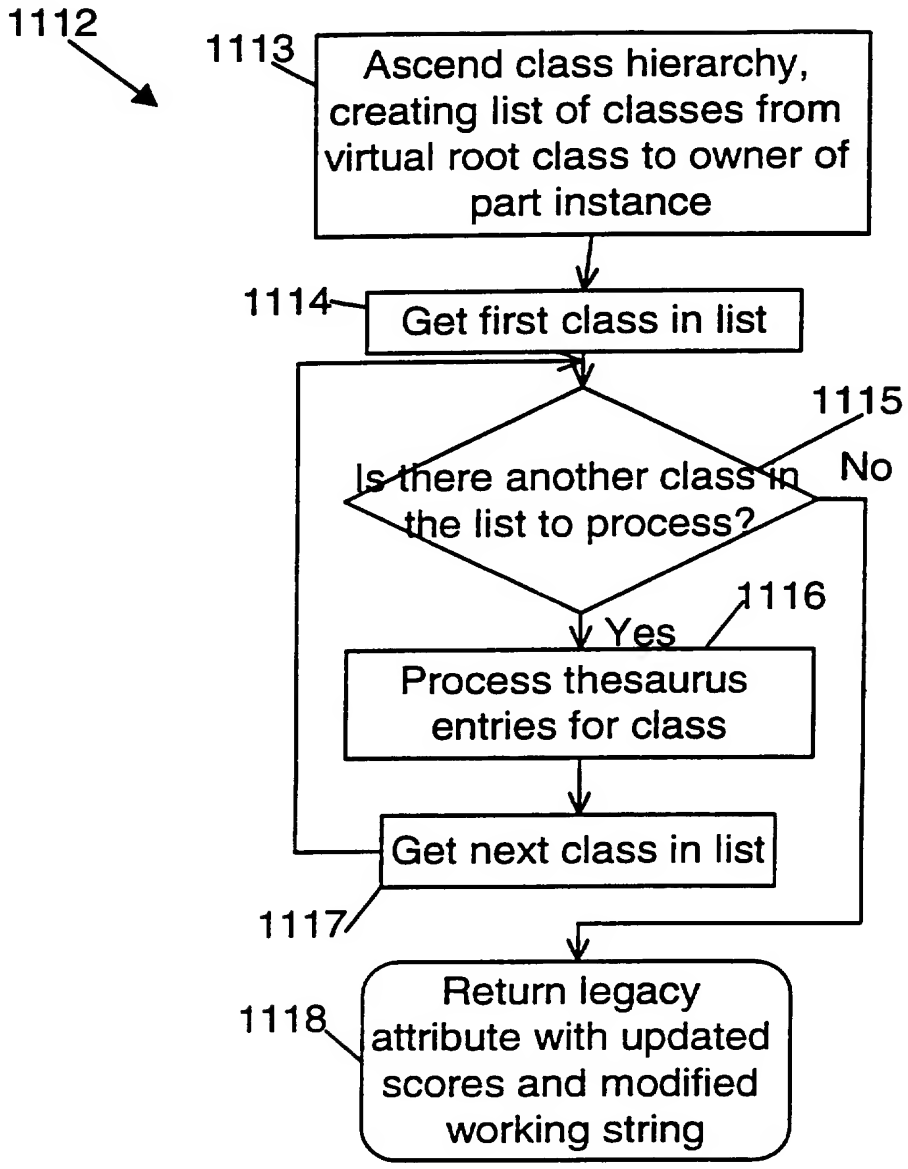

1112
1113
Ascend class hierarchy, creating list of classes from virtual root class to owner of part instance
1114
Get first class in list
1115
Is there another class in the list to process?
No
Yes
1116
Process thesaurus entries for class
Get next class in list
1117
1118
Return legacy attribute with updated scores and modified working string

FIG.135

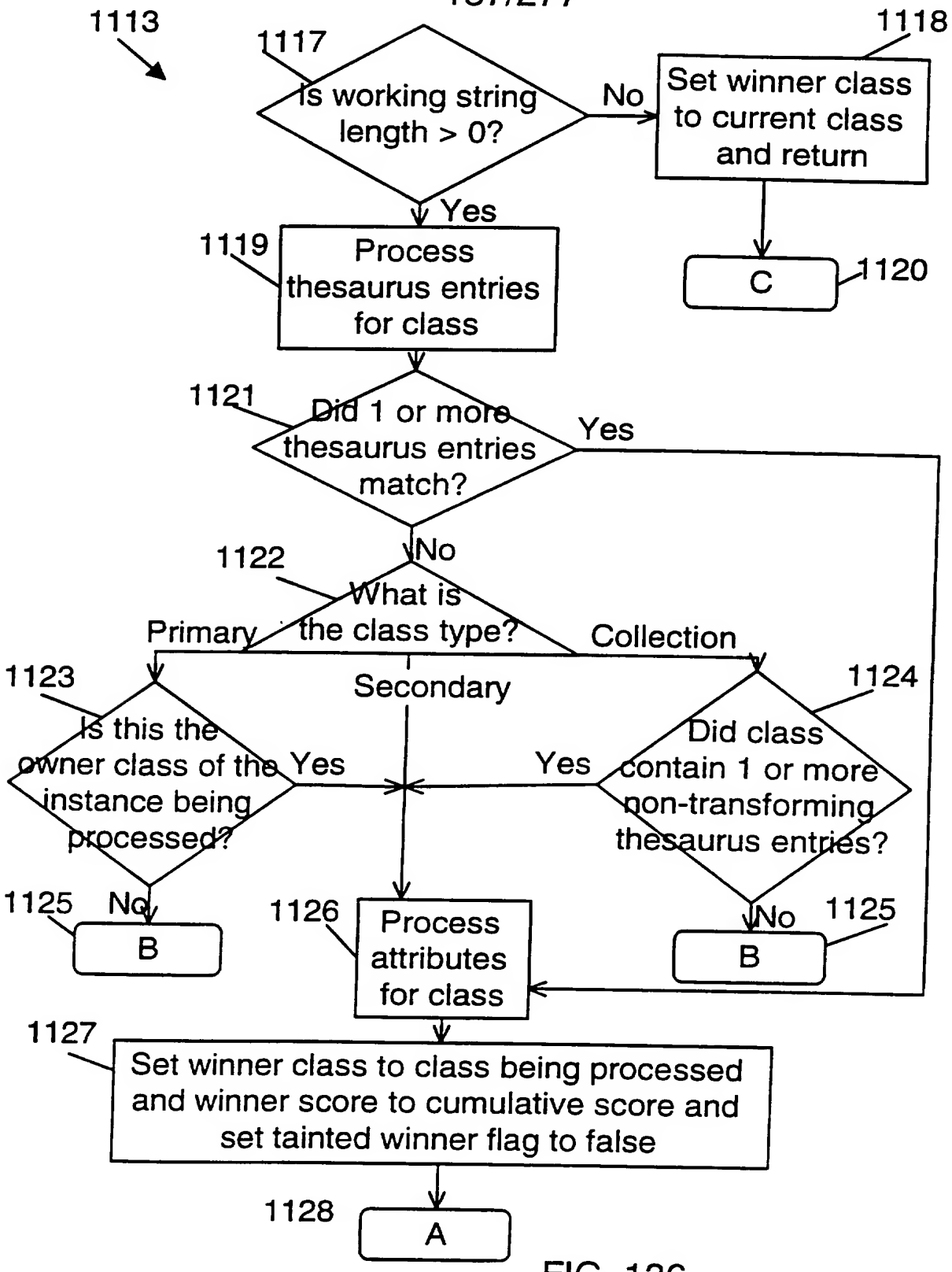

1113
1117
Is working string length > 0?
No
1118
Set winner class to current class and return
Yes
1119
Process thesaurus entries for class
C
1120
1121
Did 1 or more thesaurus entries match?
Yes
No
1122
What is the class type?
Primary
Secondary
Collection
1123
Is this the owner class of the instance being processed?
Yes
1124
Did class contain 1 or more non-transforming thesaurus entries?
Yes
1125
No
B
1126
Process attributes for class
No
1125
B
1127
Set winner class to class being processed and winner score to cumulative score and set tainted winner flag to false
1128
A

FIG. 136

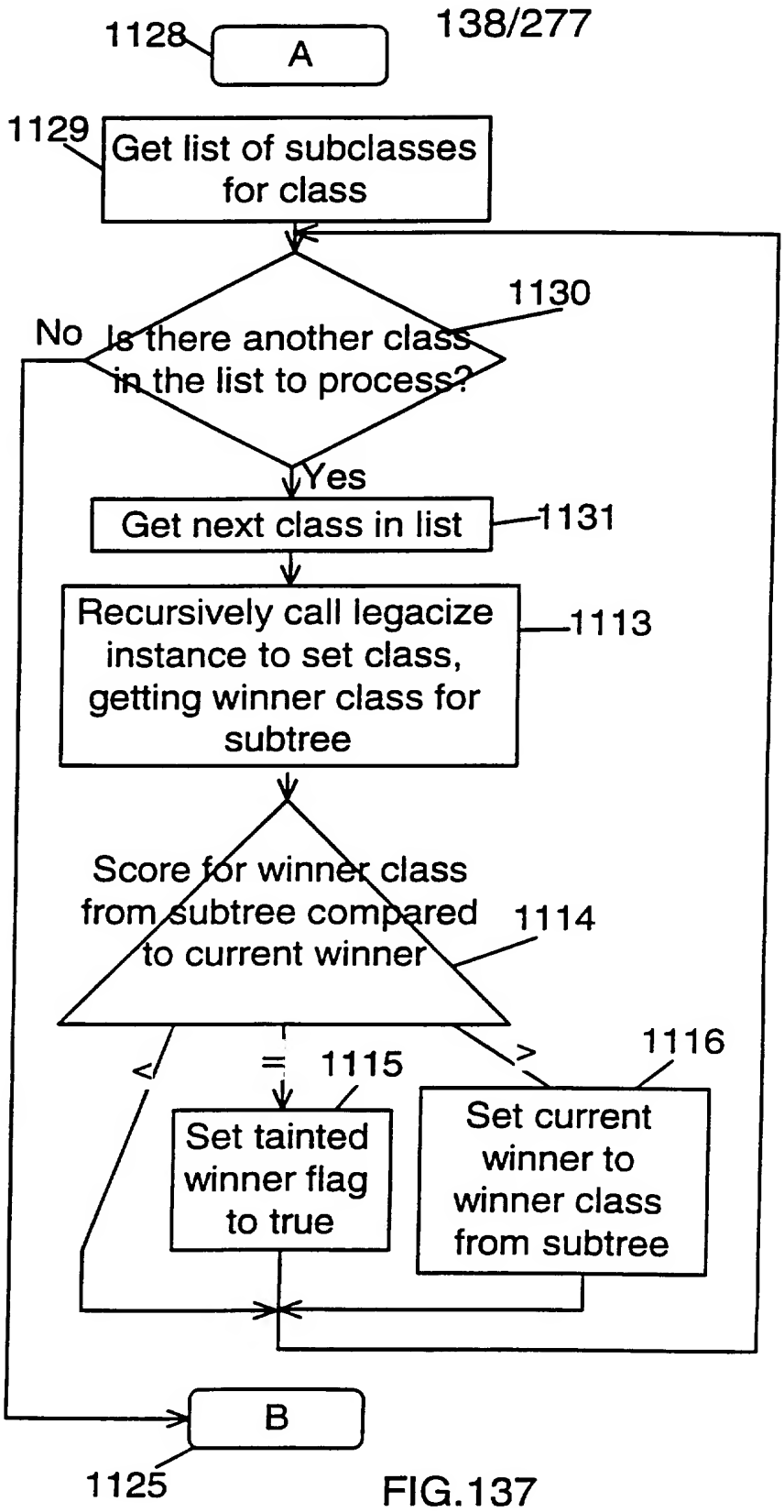

1128
A
1129
Get list of subclasses for class
1130
No
Is there another class in the list to process?
Yes
Get next class in list
1131
Recursively call legacize instance to set class, getting winner class for subtree
1113
Score for winner class from subtree compared to current winner
1114
<
=
1115
>
1116
Set tainted winner flag to true
Set current winner to winner class from subtree
B
1125

FIG.137

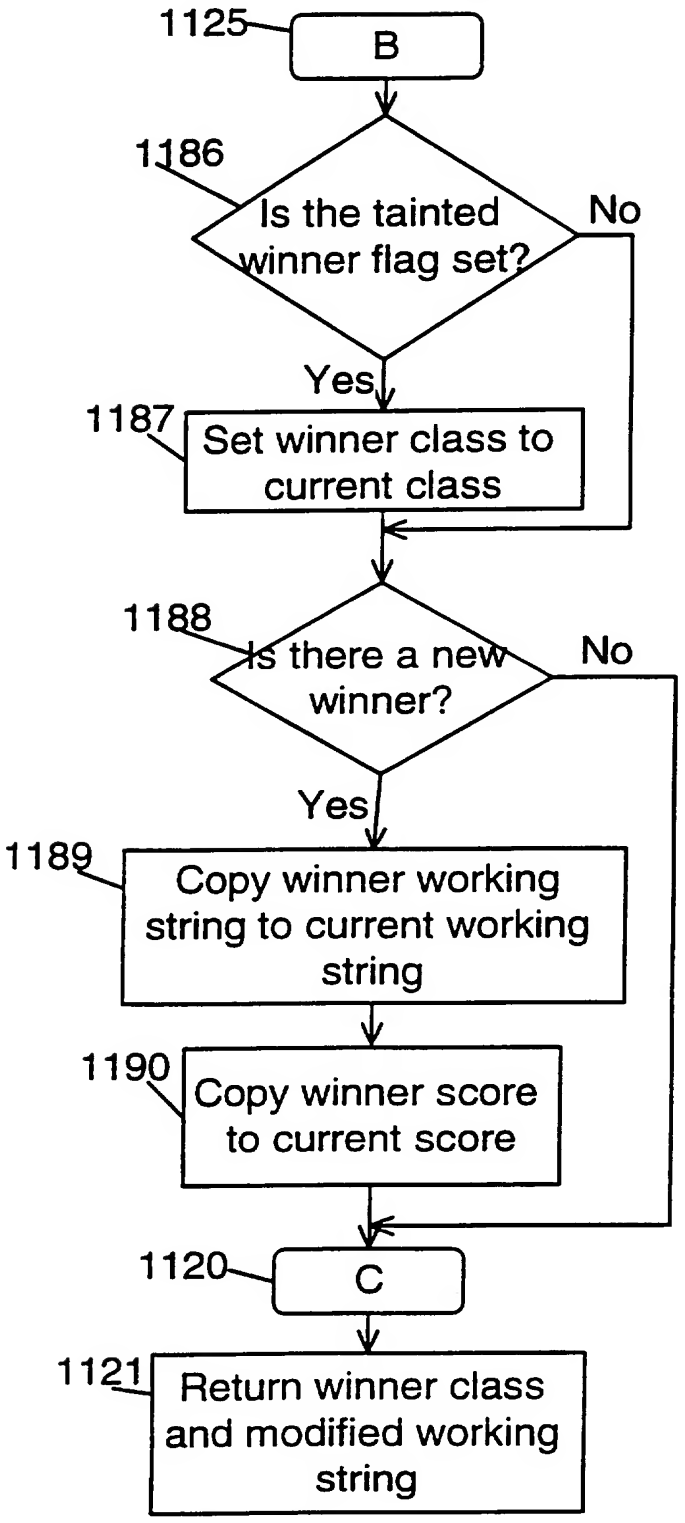

1125
B
1186
Is the tainted winner flag set?
No
Yes
1187
Set winner class to current class
1188
Is there a new winner?
No
Yes
1189
Copy winner working string to current working string
1190
Copy winner score to current score
1120
C
1121
Return winner class and modified working string

FIG. 138

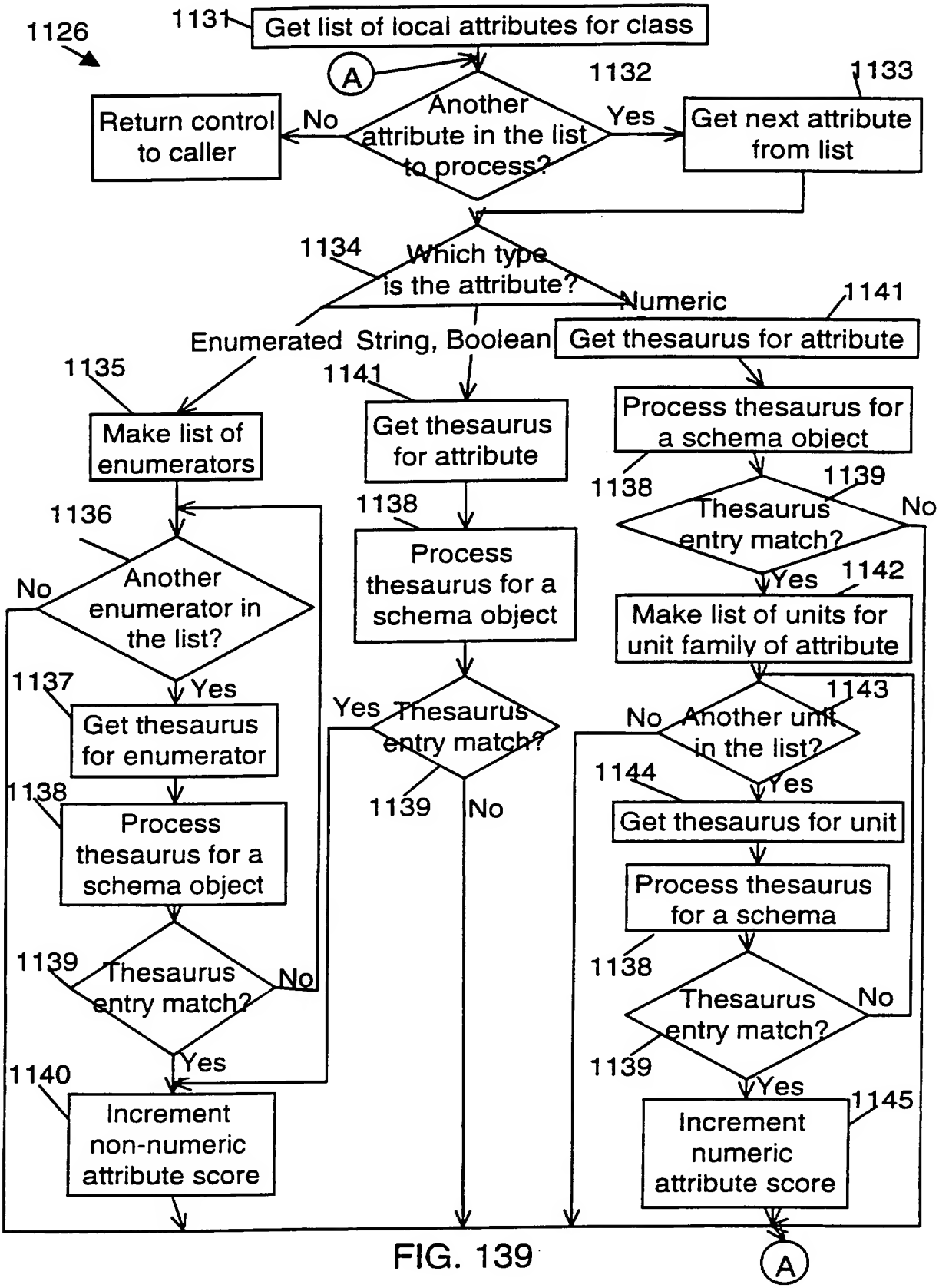

1126
1131
Get list of local attributes for class
A
1132
Another attribute in the list to process?
No
Return control to caller
Yes
1133
Get next attribute from list
1134
Which type is the attribute?
Enumerated
String, Boolean
Numeric
1141
Get thesaurus for attribute
1135
Make list of enumerators
1141
Get thesaurus for attribute
Process thesaurus for a schema object
1136
Another enumerator in the list?
No
Yes
1137
Get thesaurus for enumerator
1138
Process thesaurus for a schema object
1139
Thesaurus entry match?
No
Yes
1140
Increment non-numeric attribute score
1138
Process thesaurus for a schema object
Yes
Thesaurus entry match?
1139
No
1138
1139
Thesaurus entry match?
No
Yes
1142
Make list of units for unit family of attribute
1143
Another unit in the list?
No
Yes
1144
Get thesaurus for unit
Process thesaurus for a schema
1138
Thesaurus entry match?
No
1139
Yes
1145
Increment numeric attribute score
A

FIG. 139

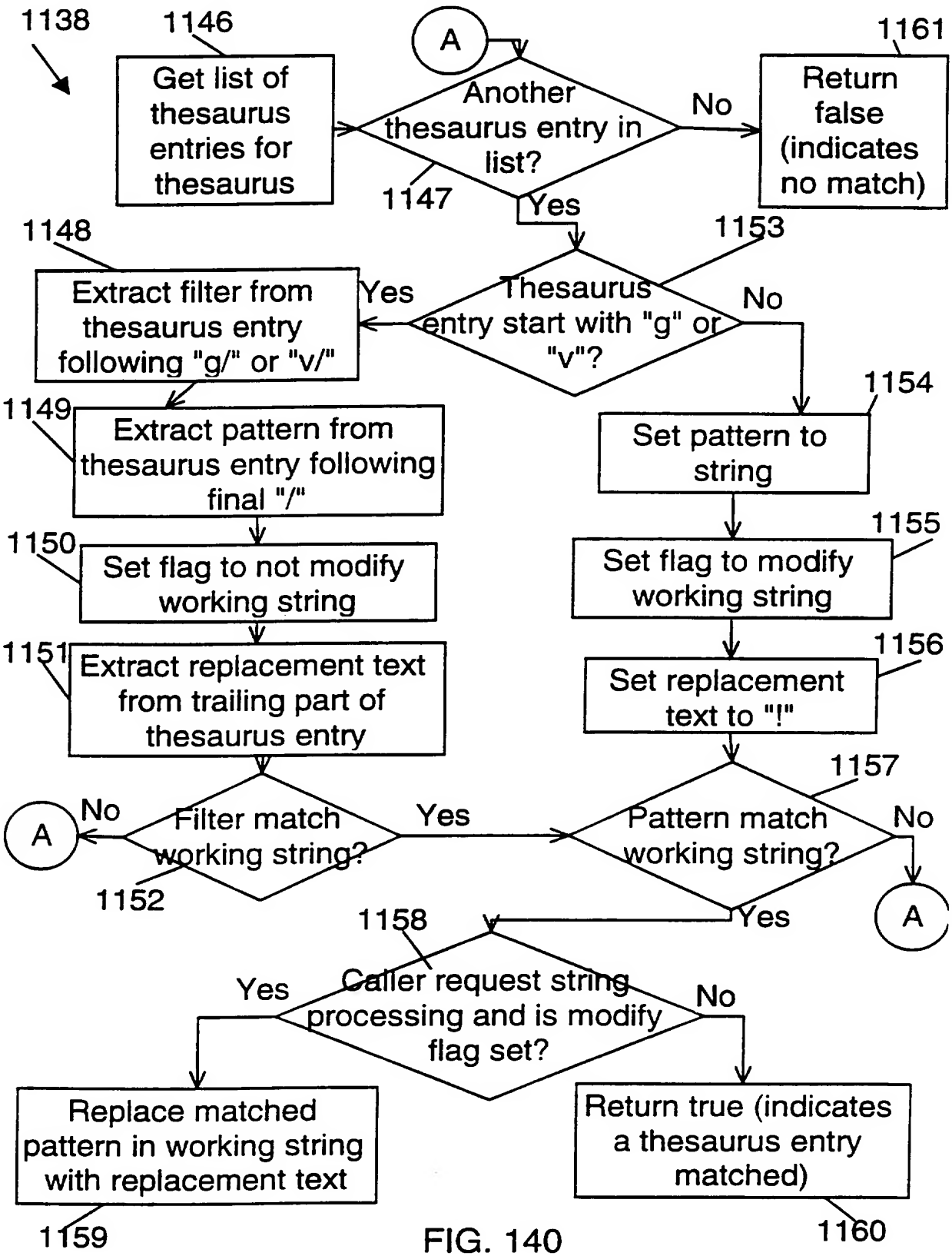

1138
1146
Get list of thesaurus entries for thesaurus
A
Another thesaurus entry in list?
1147
No
1161
Return false (indicates no match)
Yes
1148
Extract filter from thesaurus entry following "g/" or "v/"
1153
Thesaurus entry start with "g" or "v"?
Yes
No
1154
Set pattern to string
1149
Extract pattern from thesaurus entry following final "/"
1150
Set flag to not modify working string
1155
Set flag to modify working string
1151
Extract replacement text from trailing part of thesaurus entry
1156
Set replacement text to "!"
1157
A
No
Filter match working string?
1152
Yes
Pattern match working string?
No
A
Yes
1158
Caller request string processing and is modify flag set?
Yes
No
Replace matched pattern in working string with replacement text
1159
Return true (indicates a thesaurus entry matched)
1160

FIG. 140

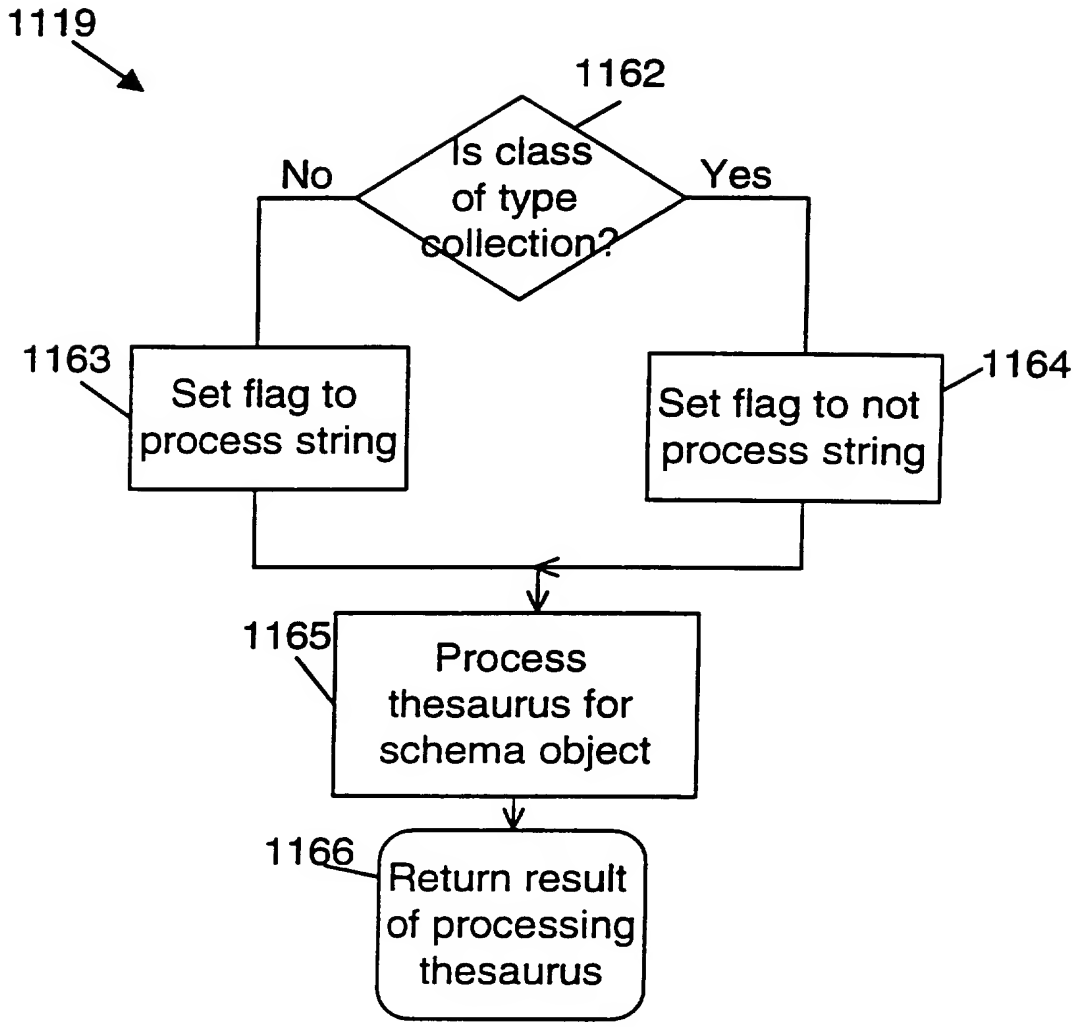

1119
1162
Is class of type collection?
No
Yes
1163
Set flag to process string
1164
Set flag to not process string
1165
Process thesaurus for schema object
1166
Return result of processing thesaurus

FIG.141

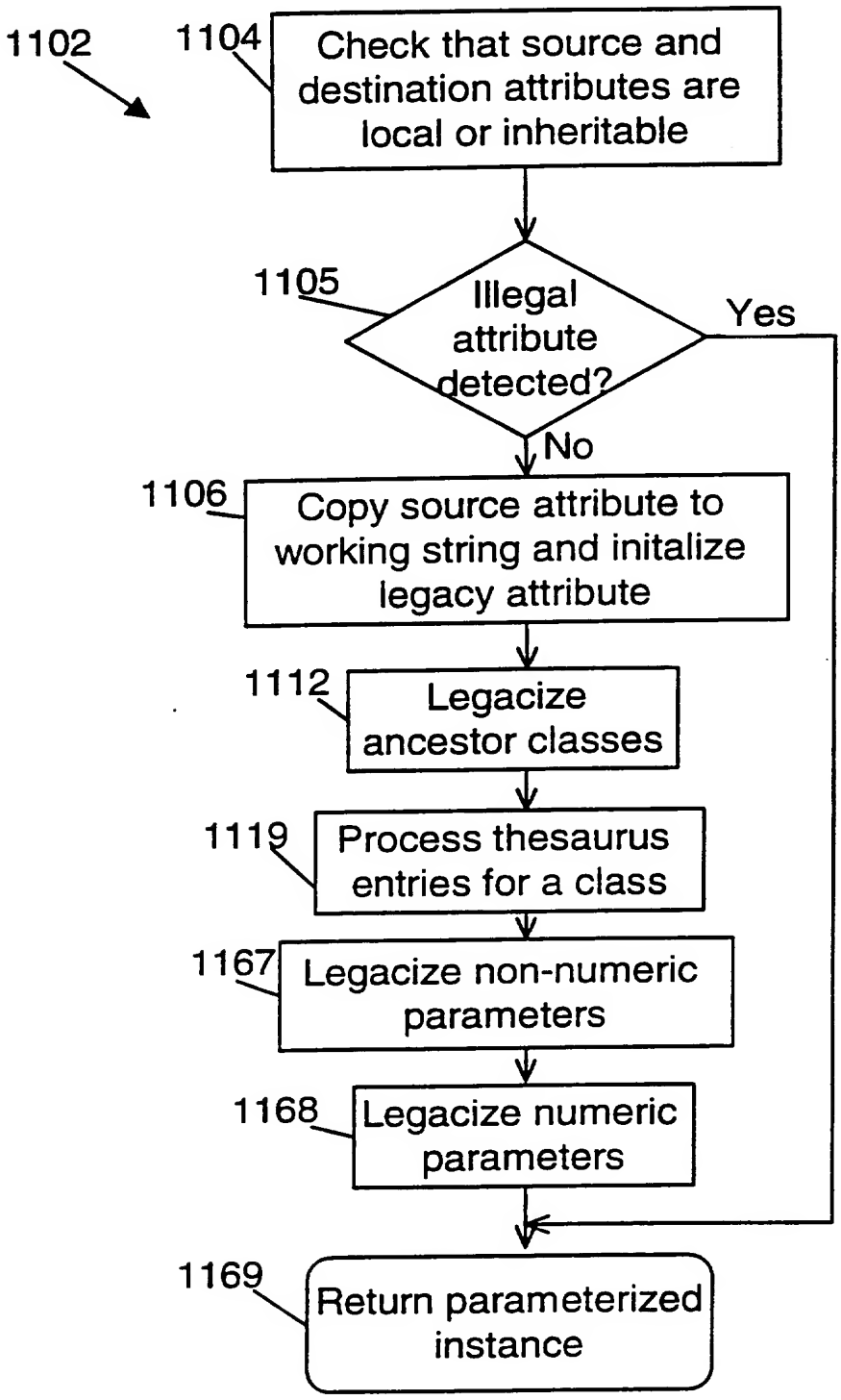

1102
1104
Check that source and destination attributes are local or inheritable
1105
Illegal attribute detected?
Yes
No
1106
Copy source attribute to working string and initalize legacy attribute
1112
Legacize ancestor classes
1119
Process thesaurus entries for a class
1167
Legacize non-numeric parameters
1168
Legacize numeric parameters
1169
Return parameterized instance

FIG. 142

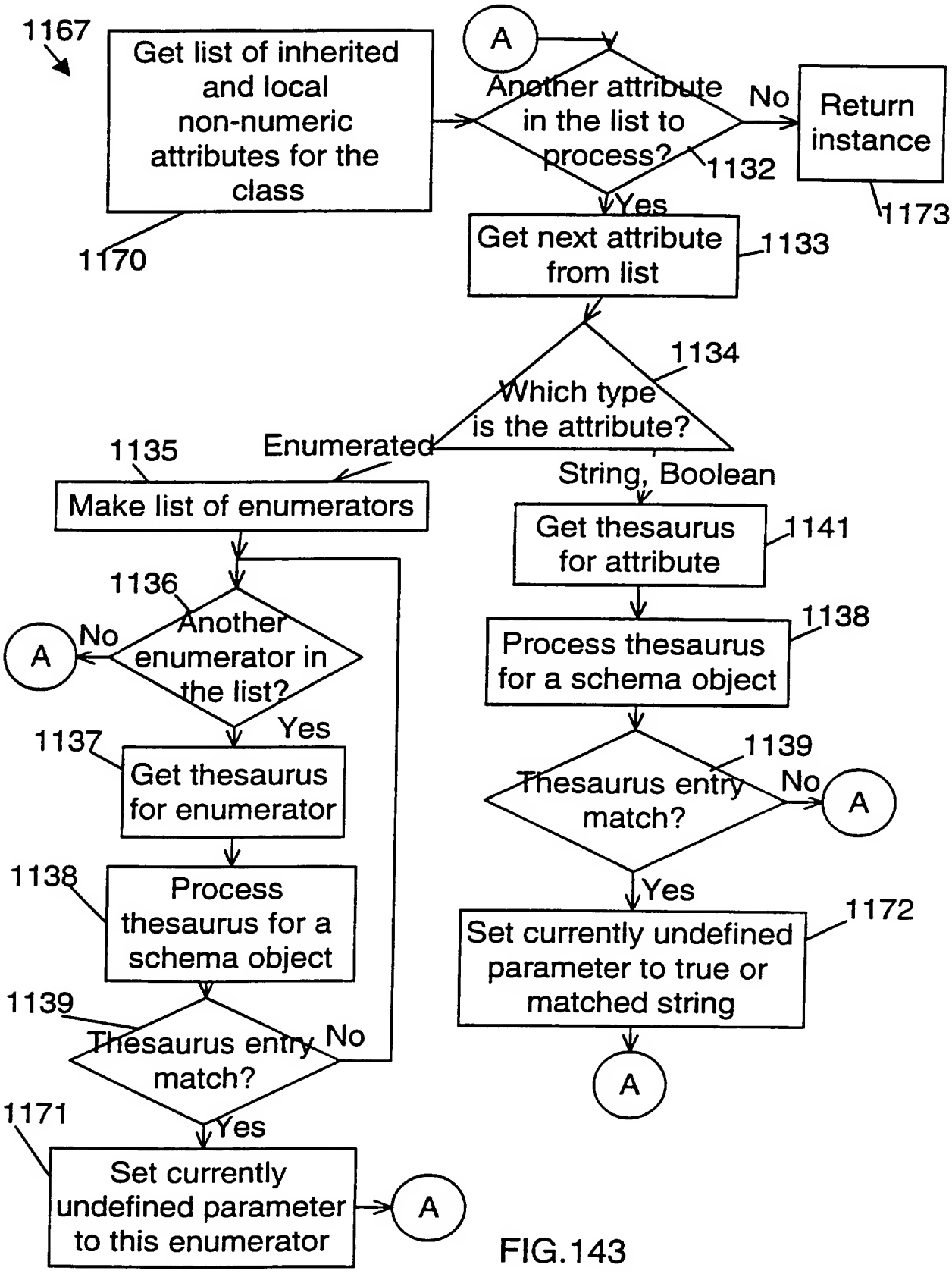

1167
Get list of inherited and local non-numeric attributes for the class
1170
A
Another attribute in the list to process?
No
Return instance
1173
1132
Yes
Get next attribute from list
1133
1134
Which type is the attribute?
Enumerated
1135
Make list of enumerators
String, Boolean
Get thesaurus for attribute
1141
1136
Another enumerator in the list?
No
A
Yes
1137
Get thesaurus for enumerator
1138
Process thesaurus for a schema object
1139
Thesaurus entry match?
No
Yes
1138
Process thesaurus for a schema object
1139
Thesaurus entry match?
No
Yes
1171
Set currently undefined parameter to this enumerator
A
1172
Set currently undefined parameter to true or matched string
A

FIG.143

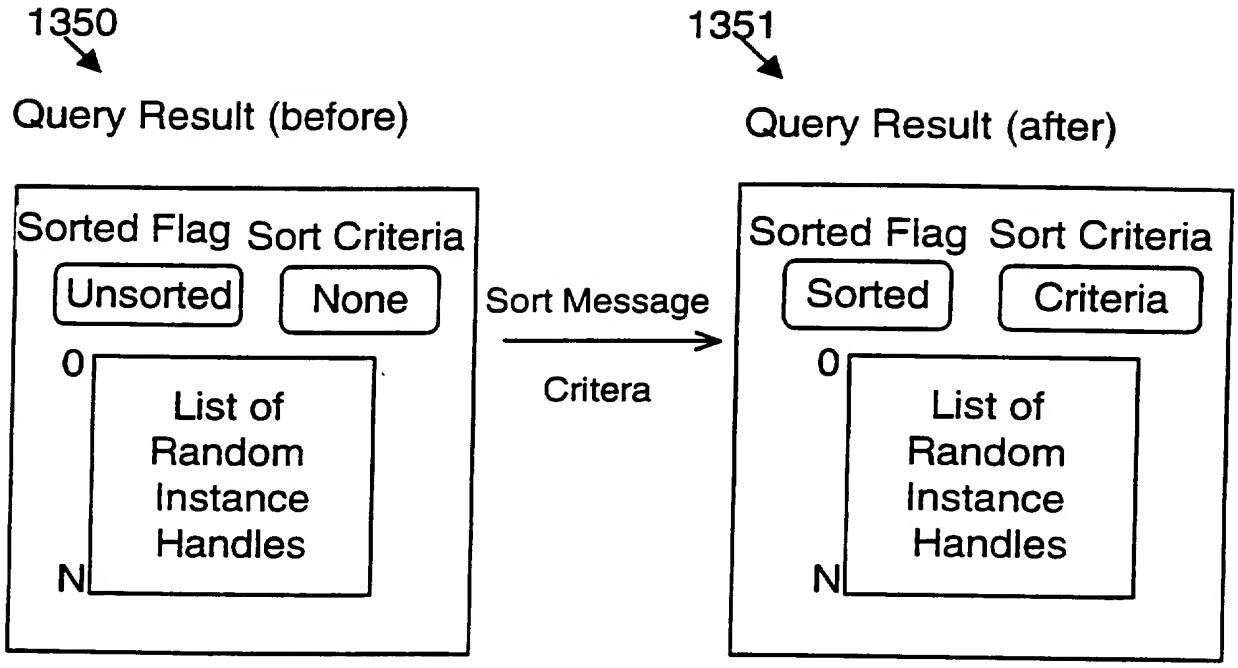

1350
Query Result (before)
Sorted Flag
Sort Criteria
Unsorted
None
0
List of Random Instance Handles
N
Sort Message
Critera
1351
Query Result (after)
Sorted Flag
Sort Criteria
Sorted
Criteria
0
List of Random Instance Handles
N

FIG. 144

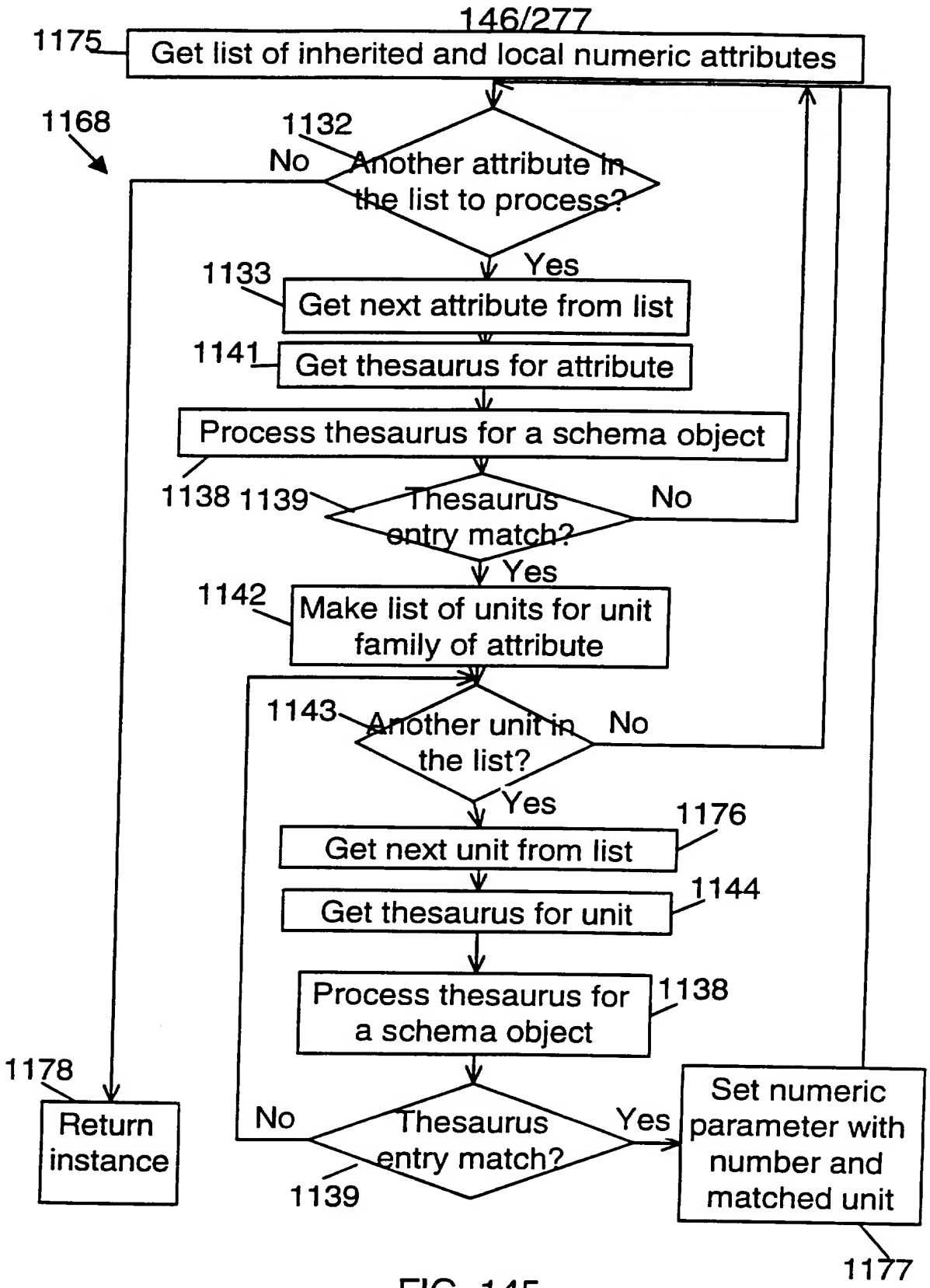

1175
Get list of inherited and local numeric attributes
1168
1132
No
Another attribute in the list to process?
Yes
1133
Get next attribute from list
1141
Get thesaurus for attribute
Process thesaurus for a schema object
1138
1139
Thesaurus entry match?
No
Yes
1142
Make list of units for unit family of attribute
1143
Another unit in the list?
No
Yes
1176
Get next unit from list
1144
Get thesaurus for unit
Process thesaurus for a schema object
1138
1178
Return instance
No
Thesaurus entry match?
Yes
1139
Set numeric parameter with number and matched unit
1177

FIG. 145

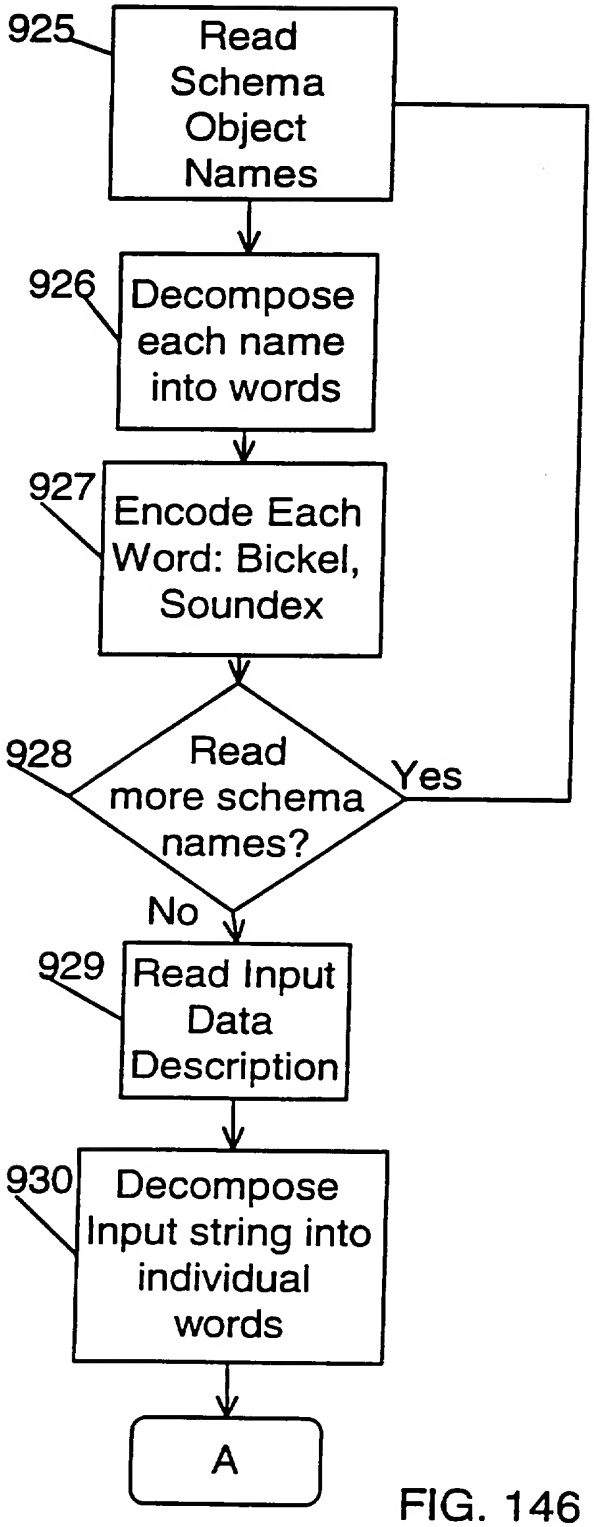

925
Read Schema Object Names
926
Decompose each name into words
927
Encode Each Word: Bickel, Soundex
928
Read more schema names?
Yes
No
929
Read Input Data Description
930
Decompose Input string into individual words
A

FIG. 146

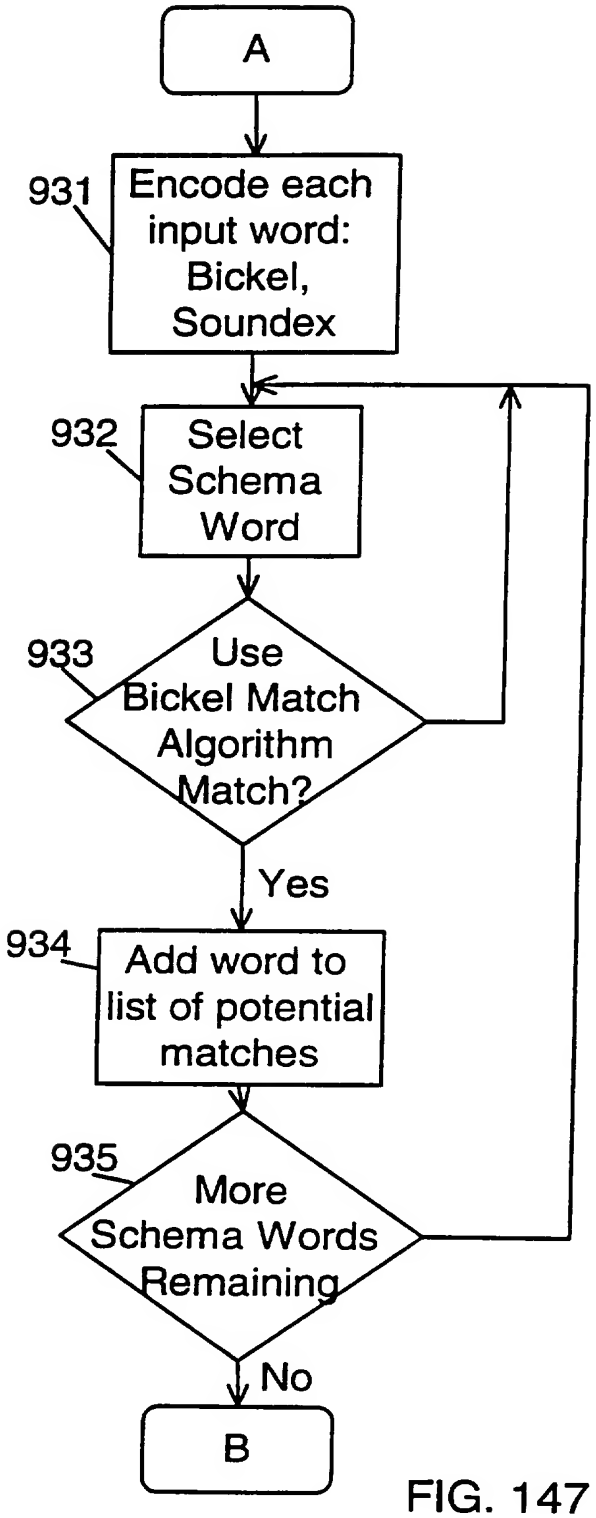

A
931
Encode each input word: Bickel, Soundex
932
Select Schema Word
933
Use Bickel Match Algorithm Match?
Yes
934
Add word to list of potential matches
935
More Schema Words Remaining
No
B

FIG. 147

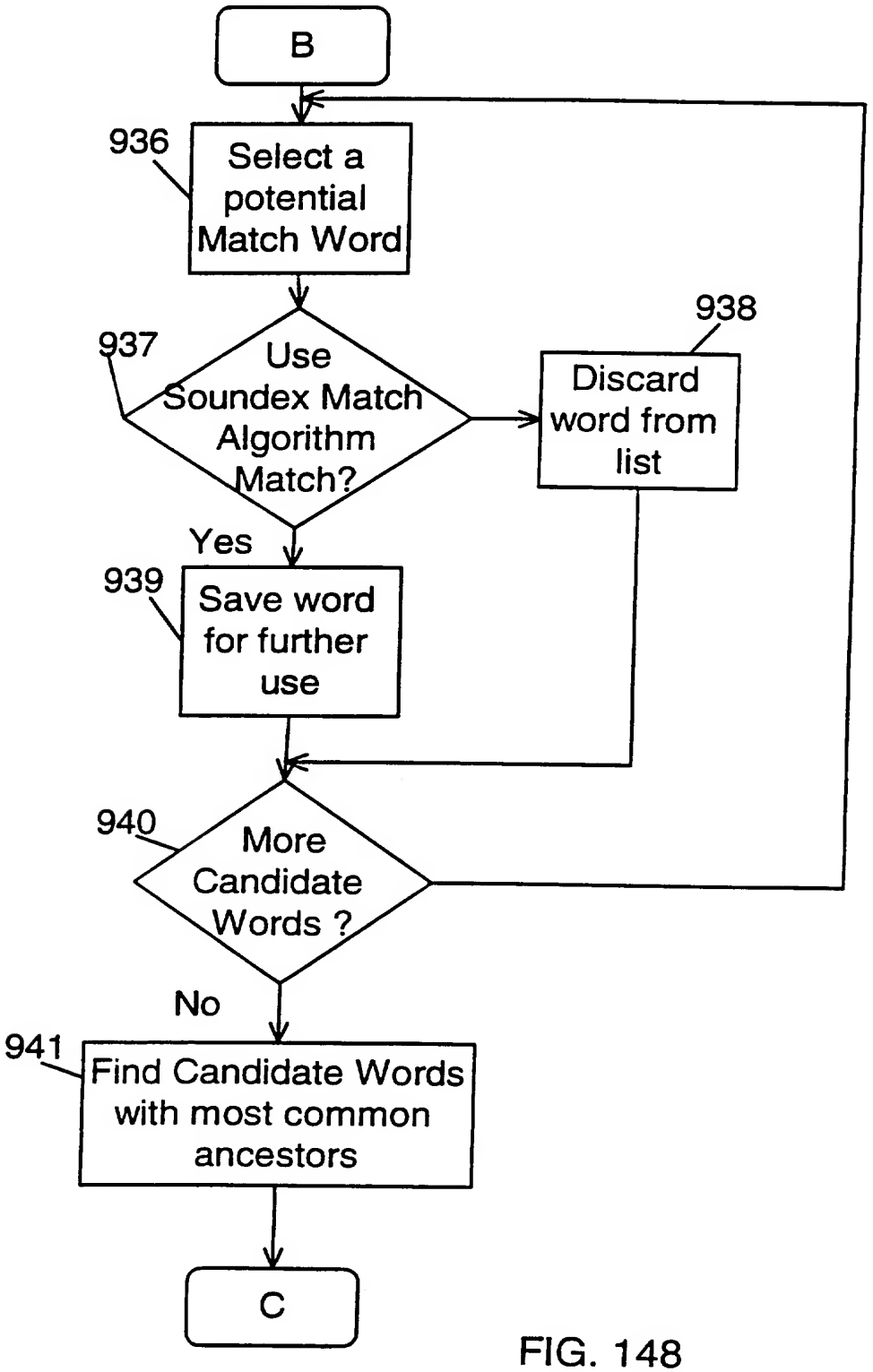

B
936
Select a potential Match Word
937
Use Soundex Match Algorithm Match?
938
Discard word from list
Yes
939
Save word for further use
940
More Candidate Words ?
No
941
Find Candidate Words with most common ancestors
C

FIG. 148

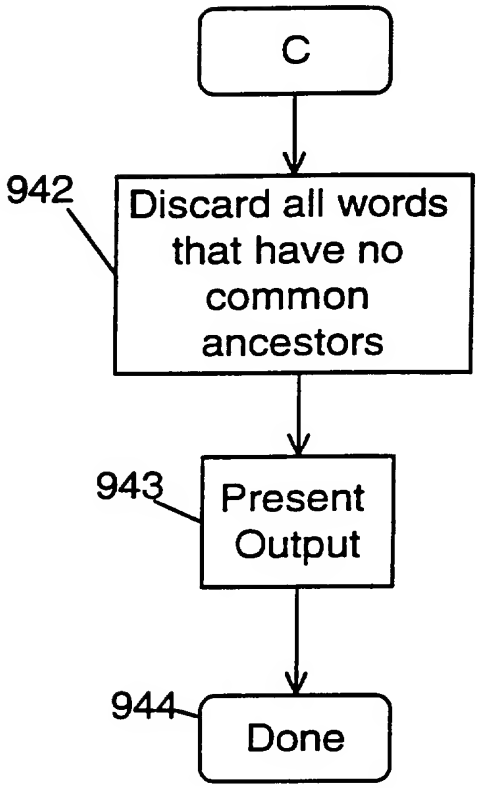

C
942
Discard all words that have no common ancestors
943
Present Output
944
Done

FIG. 149

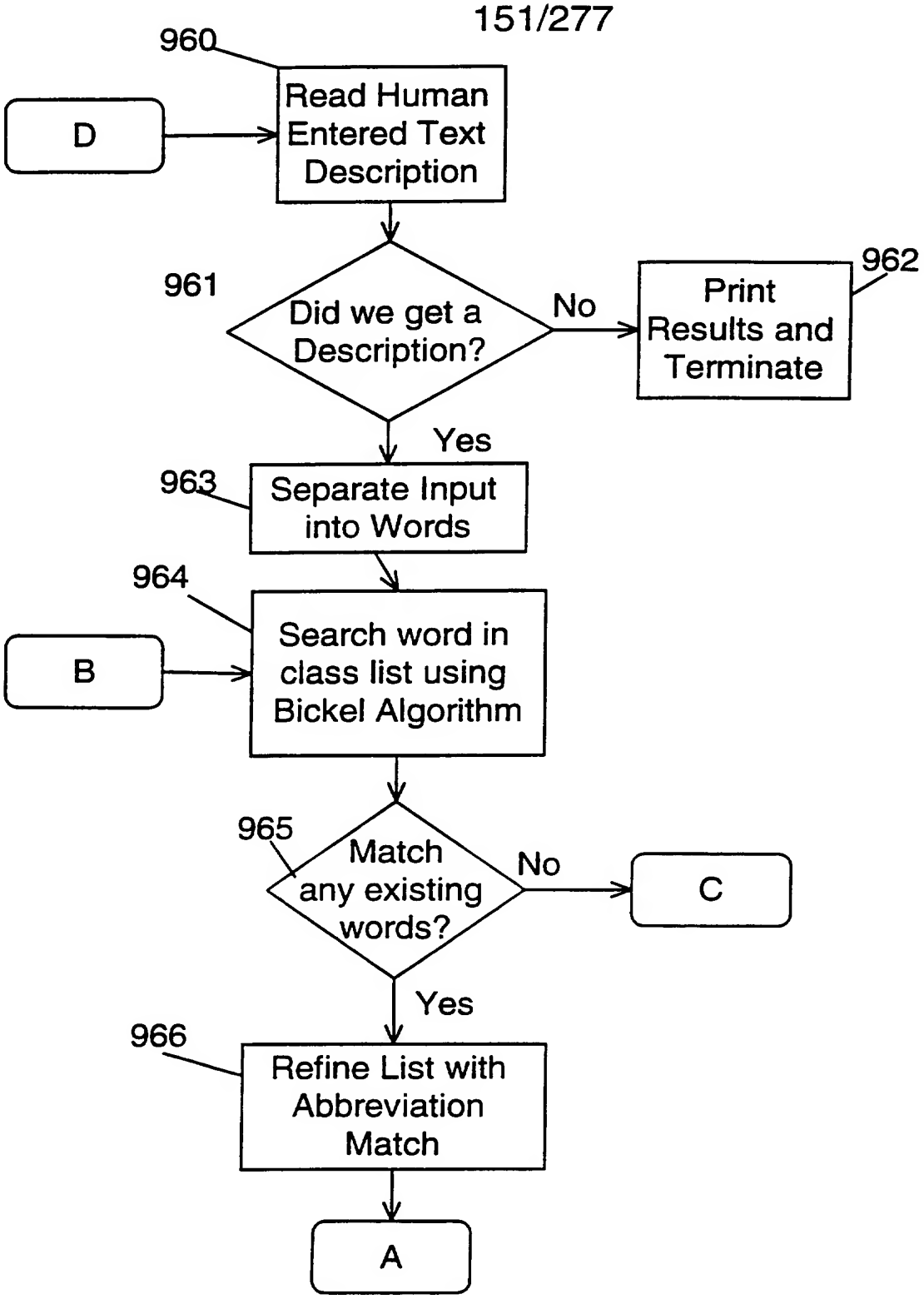

960
D
Read Human Entered Text Description
961
Did we get a Description?
No
962
Print Results and Terminate
Yes
963
Separate Input into Words
964
B
Search word in class list using Bickel Algorithm
965
Match any existing words?
No
C
Yes
966
Refine List with Abbreviation Match
A

FIG. 150

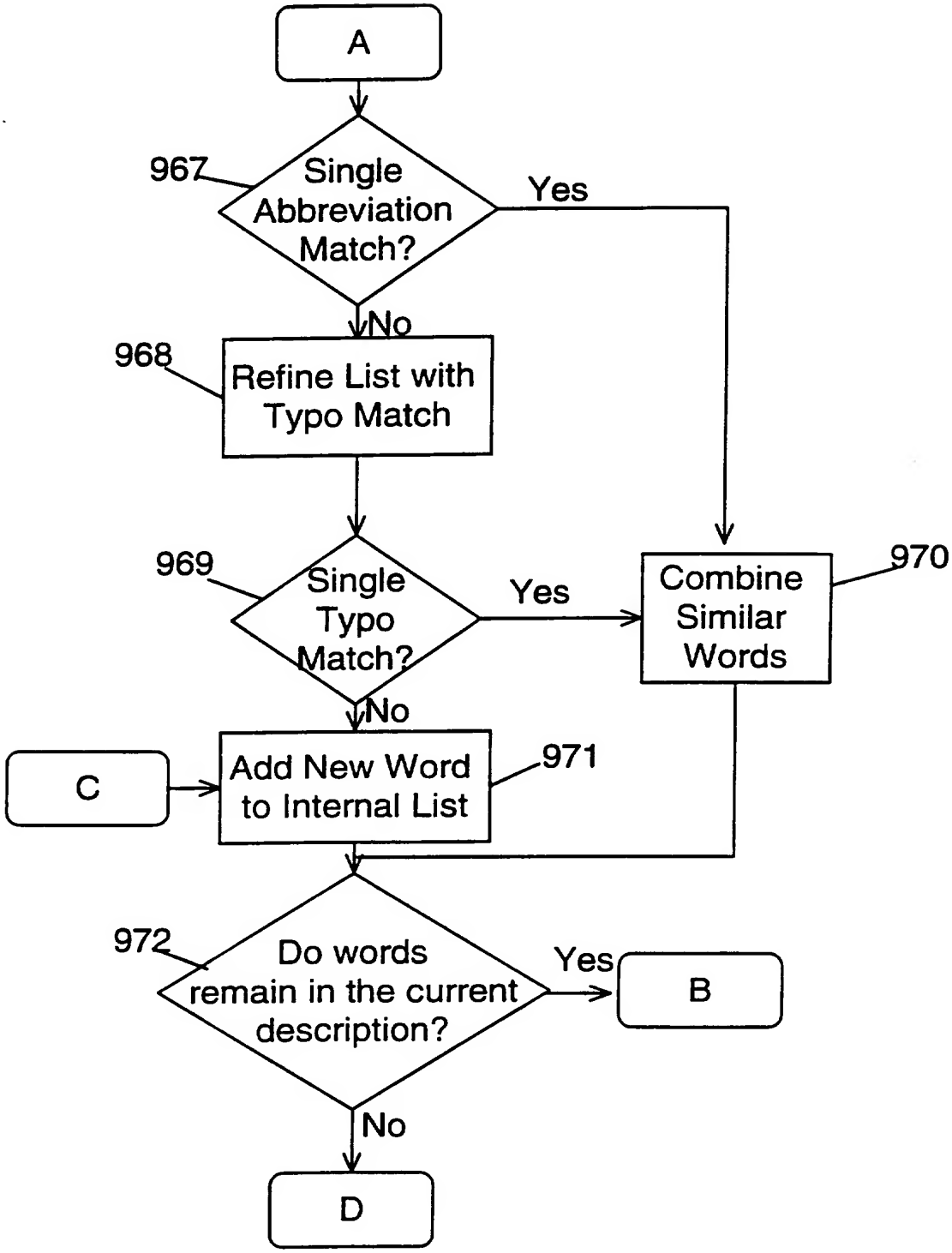

A
967
Single Abbreviation Match?
Yes
No
968
Refine List with Typo Match
969
Single Typo Match?
Yes
970
Combine Similar Words
No
C
Add New Word to Internal List
971
972
Do words remain in the current description?
Yes
B
No
D

FIG. 151

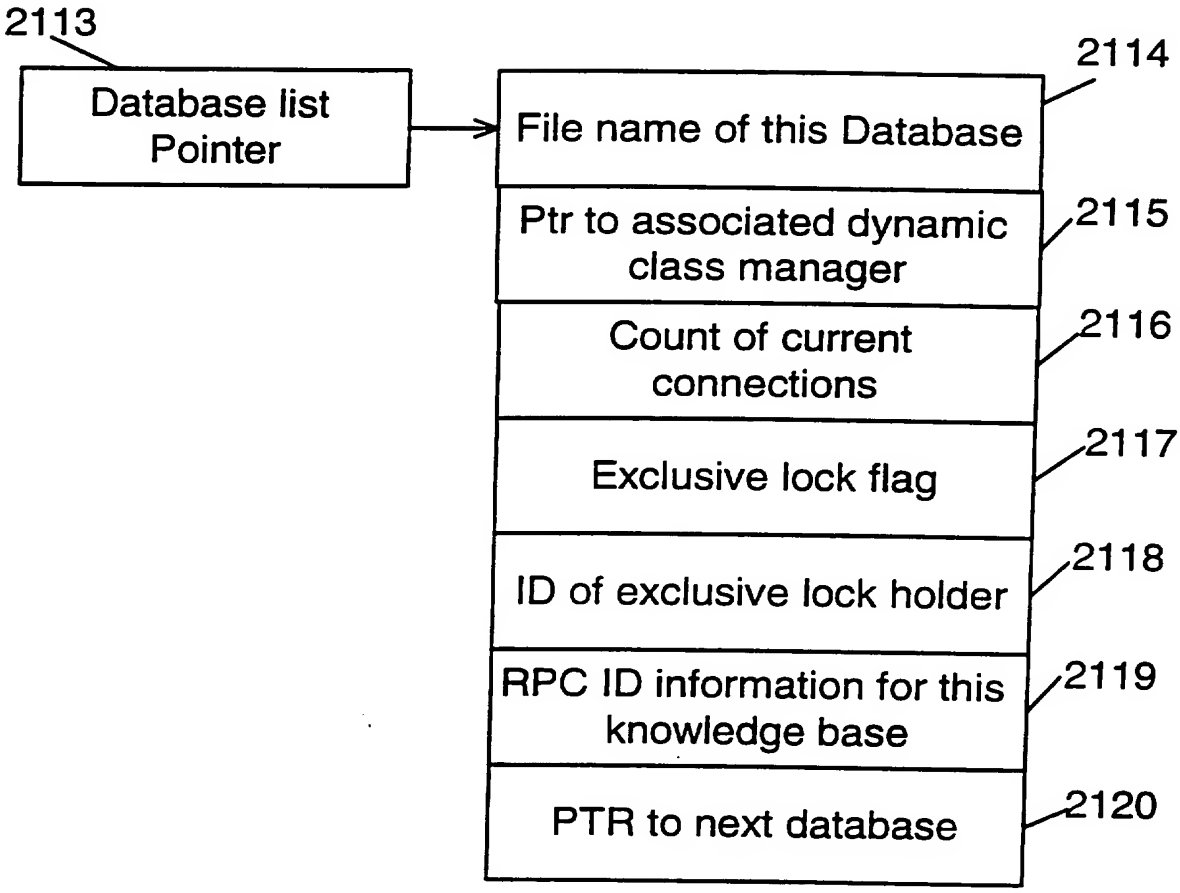

2113
Database list Pointer
2114
File name of this Database
2115
Ptr to associated dynamic class manager
2116
Count of current connections
2117
Exclusive lock flag
2118
ID of exclusive lock holder
2119
RPC ID information for this knowledge base
2120
PTR to next database

FIG. 152

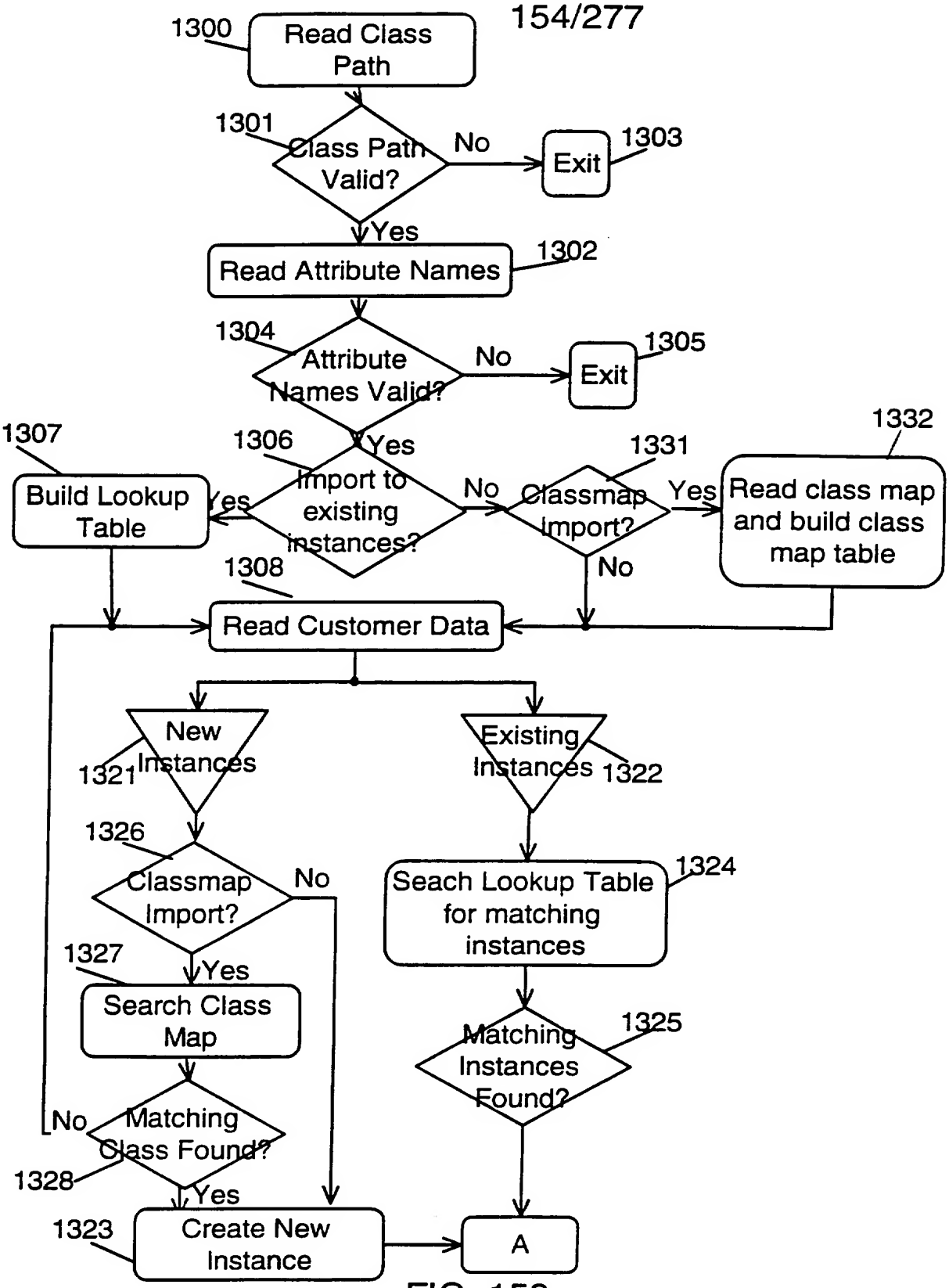

1300
Read Class Path
1301
Class Path Valid?
No
Exit
1303
Yes
Read Attribute Names
1302
1304
Attribute Names Valid?
No
Exit
1305
Yes
1307
Build Lookup Table
1306
Import to existing instances?
Yes
No
1331
Classmap Import?
Yes
1332
Read class map and build class map table
No
1308
Read Customer Data
New Instances
1321
Existing Instances
1322
1326
Classmap Import?
No
1327
Yes
Search Class Map
Matching Class Found?
No
1328
Yes
1323
Create New Instance
Seach Lookup Table for matching instances
1324
Matching Instances Found?
1325
A

FIG. 153

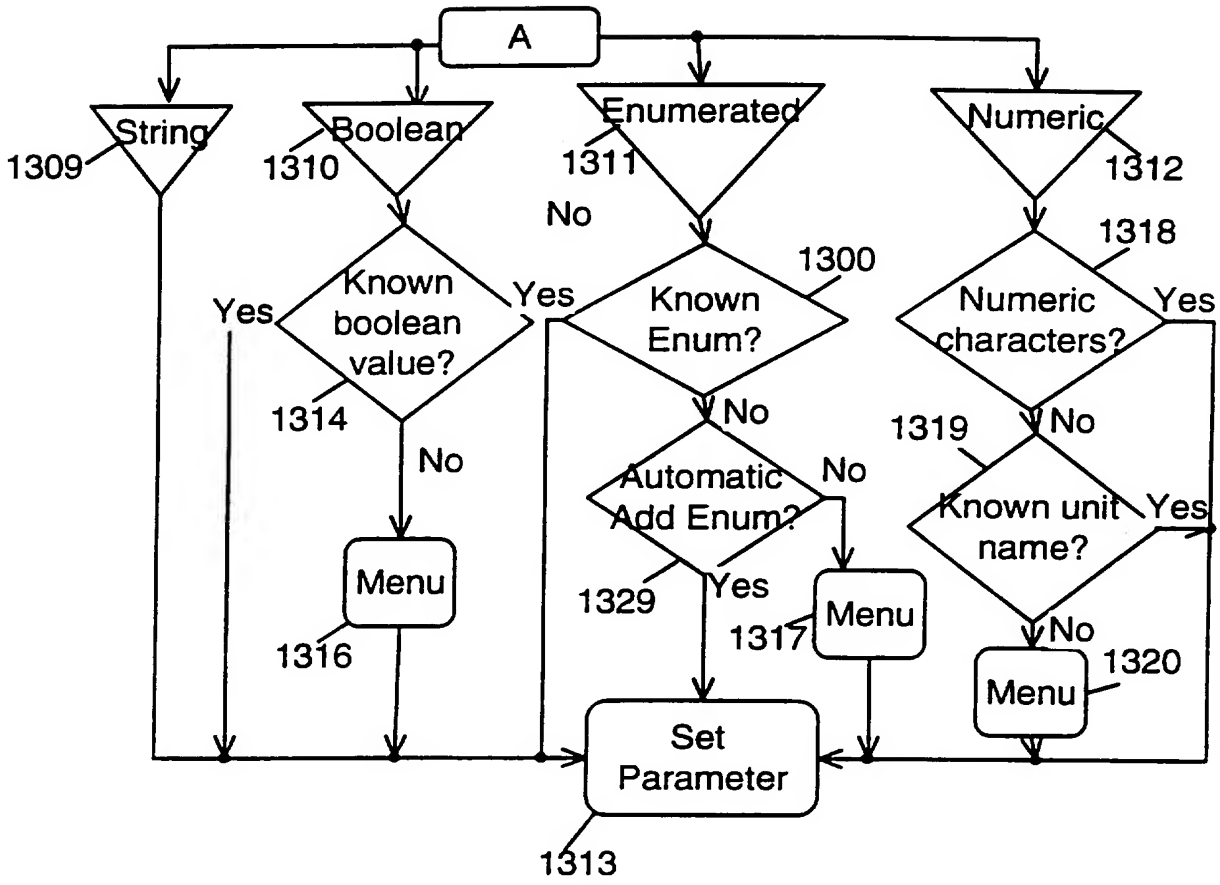

A
String
1309
Boolean
1310
Enumerated
1311
Numeric
1312
No
1300
Known Enum?
Yes
Known boolean value?
Yes
1314
No
Menu
1316
No
Automatic Add Enum?
No
1329
Yes
Menu
1317
1318
Numeric characters?
Yes
1319
No
Known unit name?
Yes
No
Menu
1320
Set Parameter
1313

FIG. 154

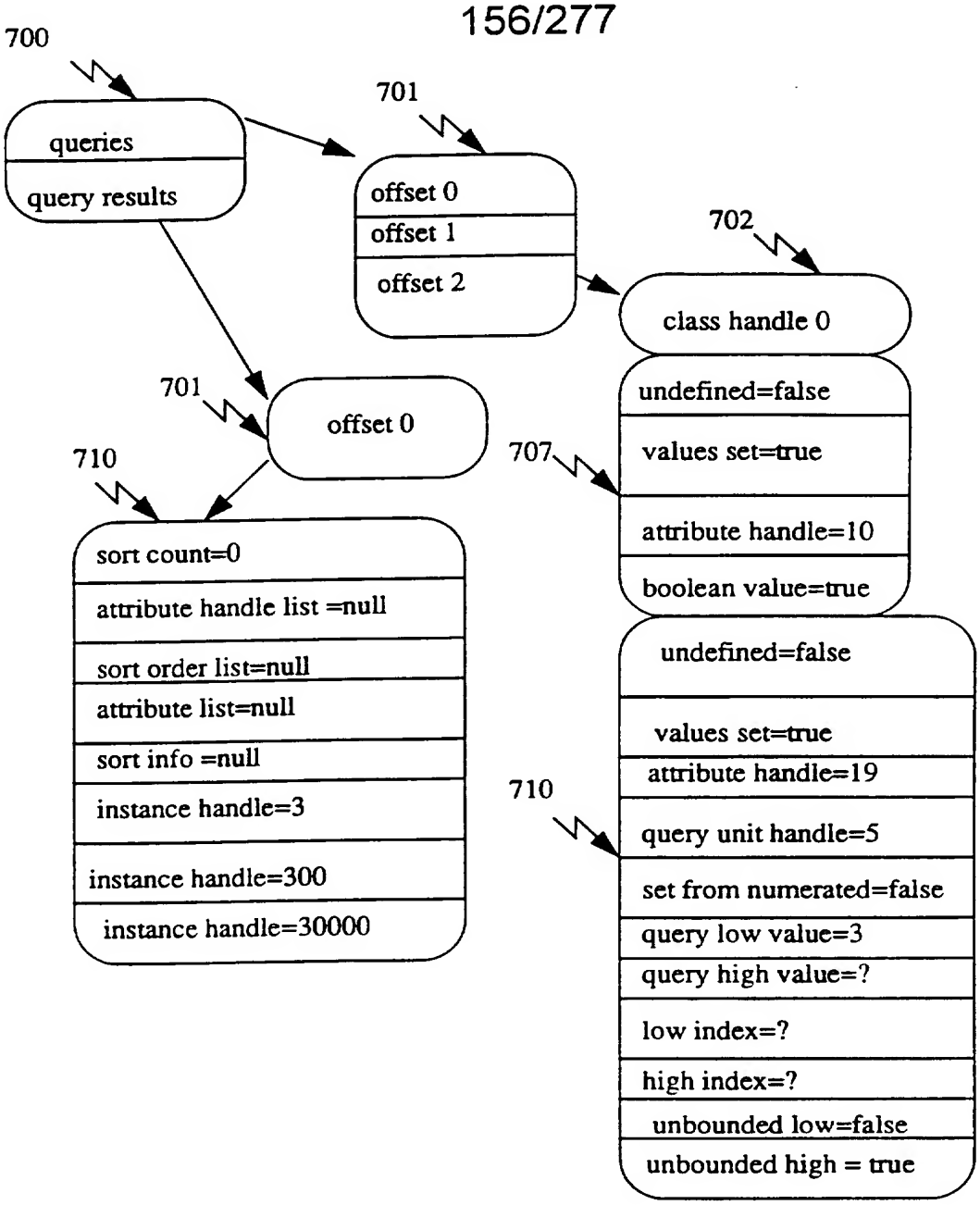

700
queries
query results
701
offset 0
offset 1
offset 2
702
class handle 0
undefined=false
values set=true
attribute handle=10
boolean value=true
707
701
offset 0
710
sort count=0
attribute handle list =null
sort order list=null
attribute list=null
sort info =null
instance handle=3
instance handle=300
instance handle=30000
undefined=false
values set=true
attribute handle=19
710
query unit handle=5
set from numerated=false
query low value=3
query high value=?
low index=?
high index=?
unbounded low=false
unbounded high = true

FIG. 155

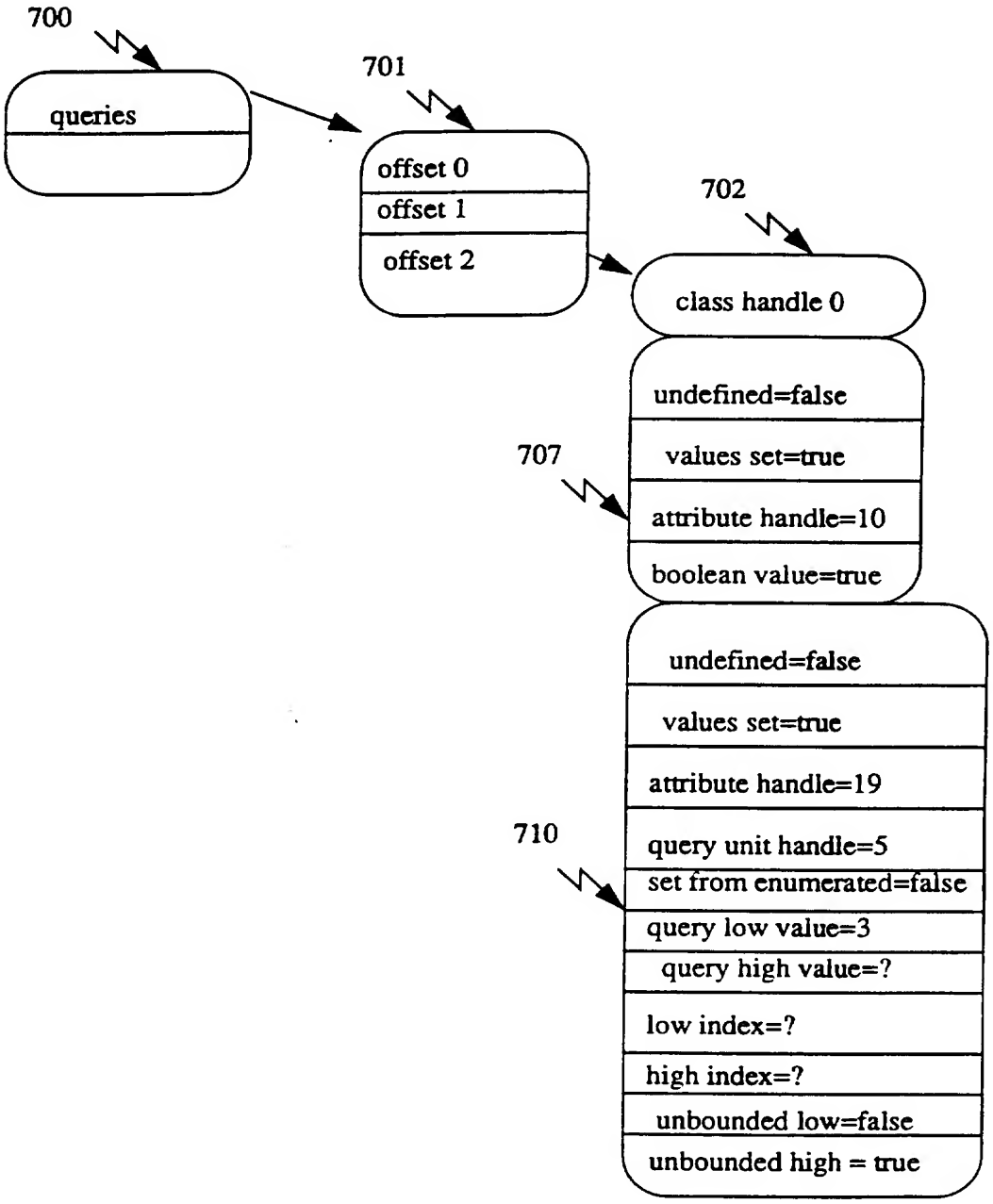

700
queries
701
offset 0
offset 1
offset 2
702
class handle 0
undefined=false
values set=true
attribute handle=10
boolean value=true
707
undefined=false
values set=true
attribute handle=19
query unit handle=5
set from enumerated=false
query low value=3
query high value=?
low index=?
high index=?
unbounded low=false
unbounded high = true
710

FIG. 156

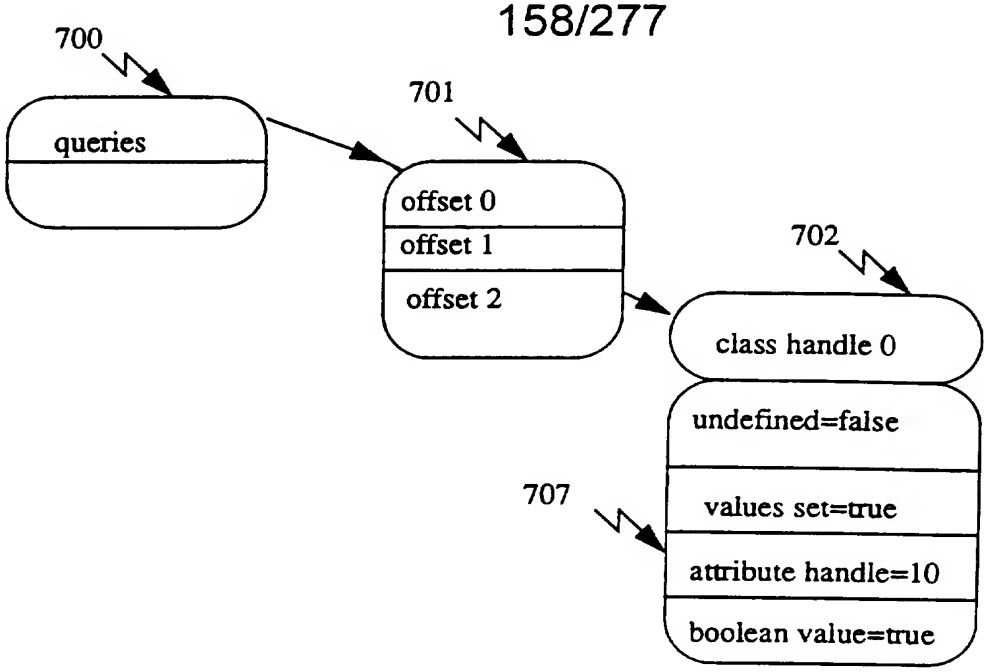

700
queries
701
offset 0
offset 1
offset 2
702
class handle 0
undefined=false
707
values set=true
attribute handle=10
boolean value=true

FIG. 157

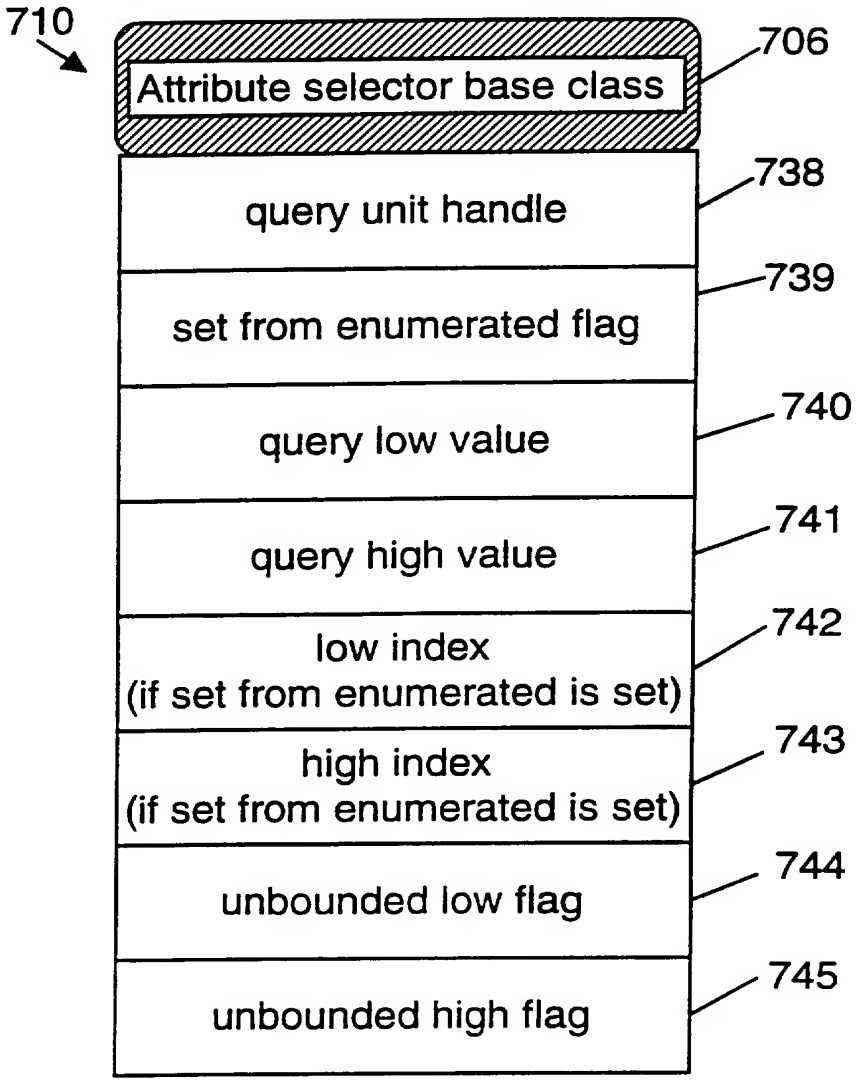

710
Attribute selector base class
706
query unit handle
738
set from enumerated flag
739
query low value
740
query high value
741
low index
(if set from enumerated is set)
742
high index
(if set from enumerated is set)
743
unbounded low flag
744
unbounded high flag
745

FIG. 158

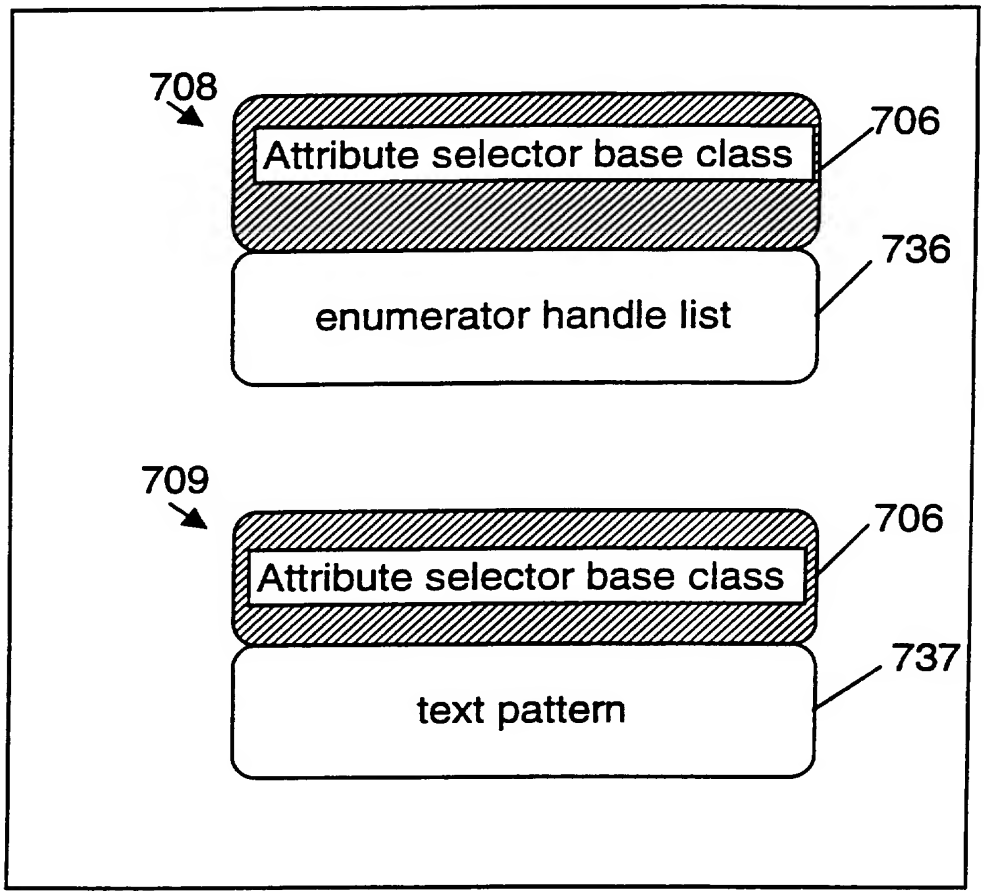

708
Attribute selector base class
706
736
enumerator handle list
709
Attribute selector base class
706
737
text pattern

FIG. 159

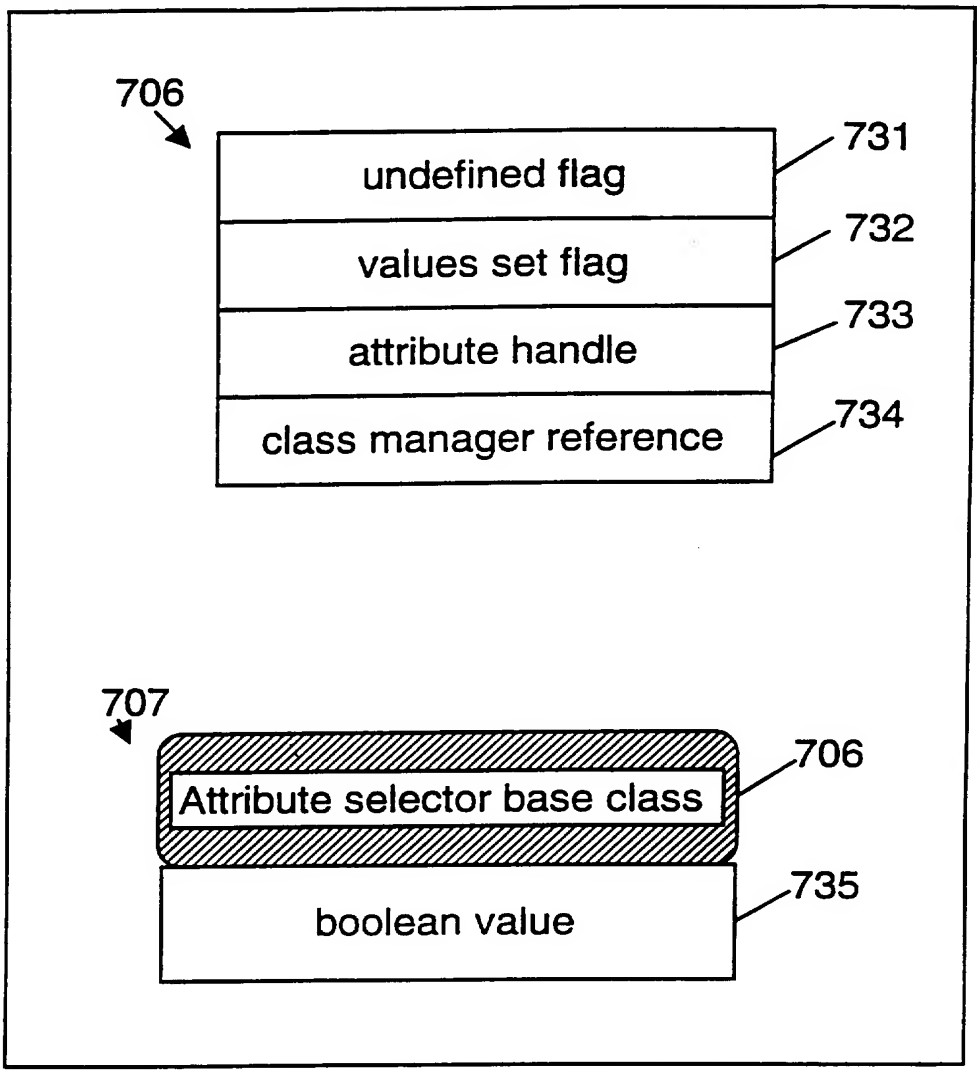

706
undefined flag
731
values set flag
732
attribute handle
733
class manager reference
734
707
Attribute selector base class
706
boolean value
735

FIG. 160

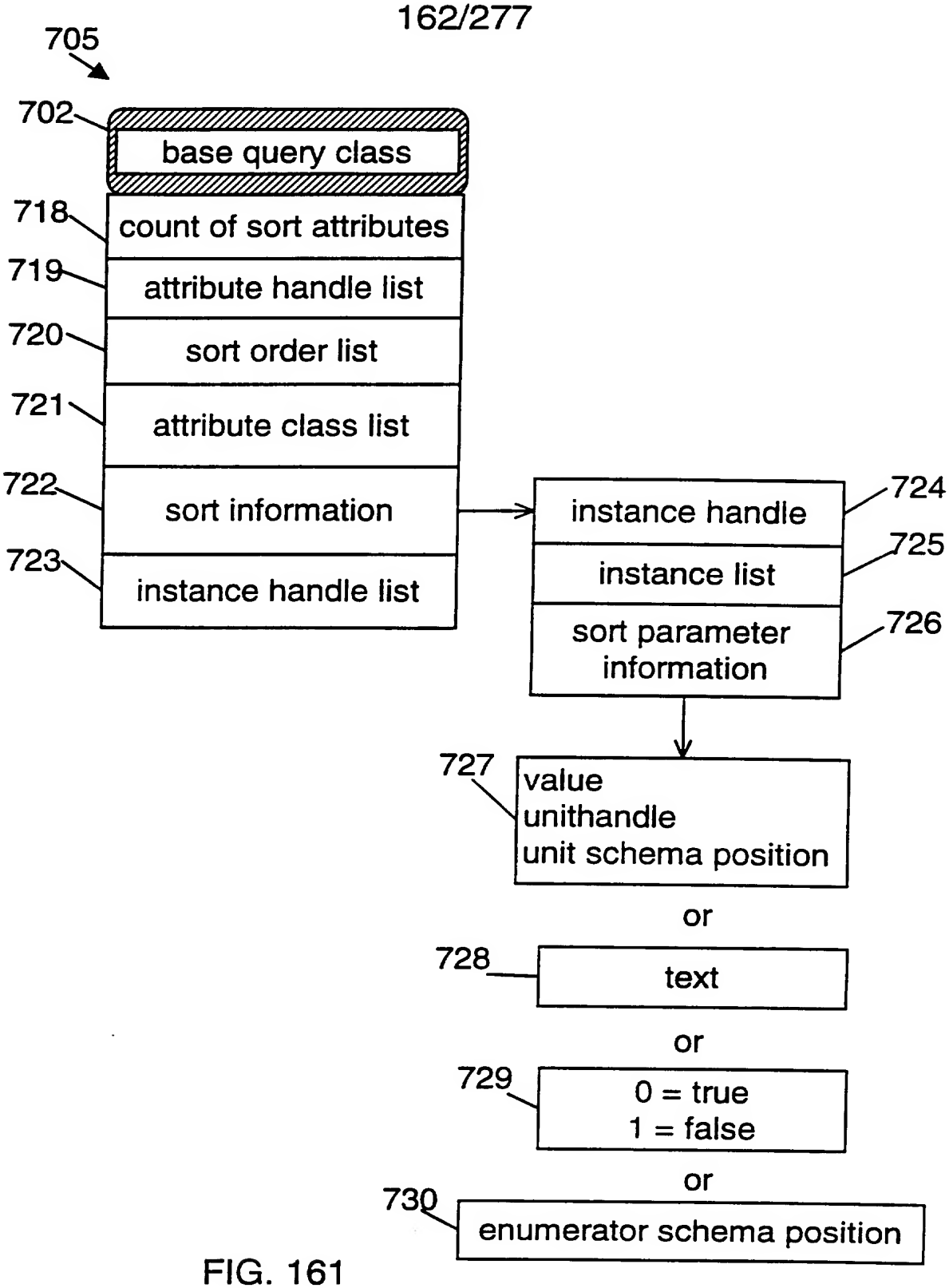

705
702
base query class
718
count of sort attributes
719
attribute handle list
720
sort order list
721
attribute class list
722
sort information
723
instance handle list
724
instance handle
725
instance list
726
sort parameter information
727
value
unithandle
unit schema position
or
728
text
or
729
0 = true
1 = false
or
730
enumerator schema position

FIG. 161

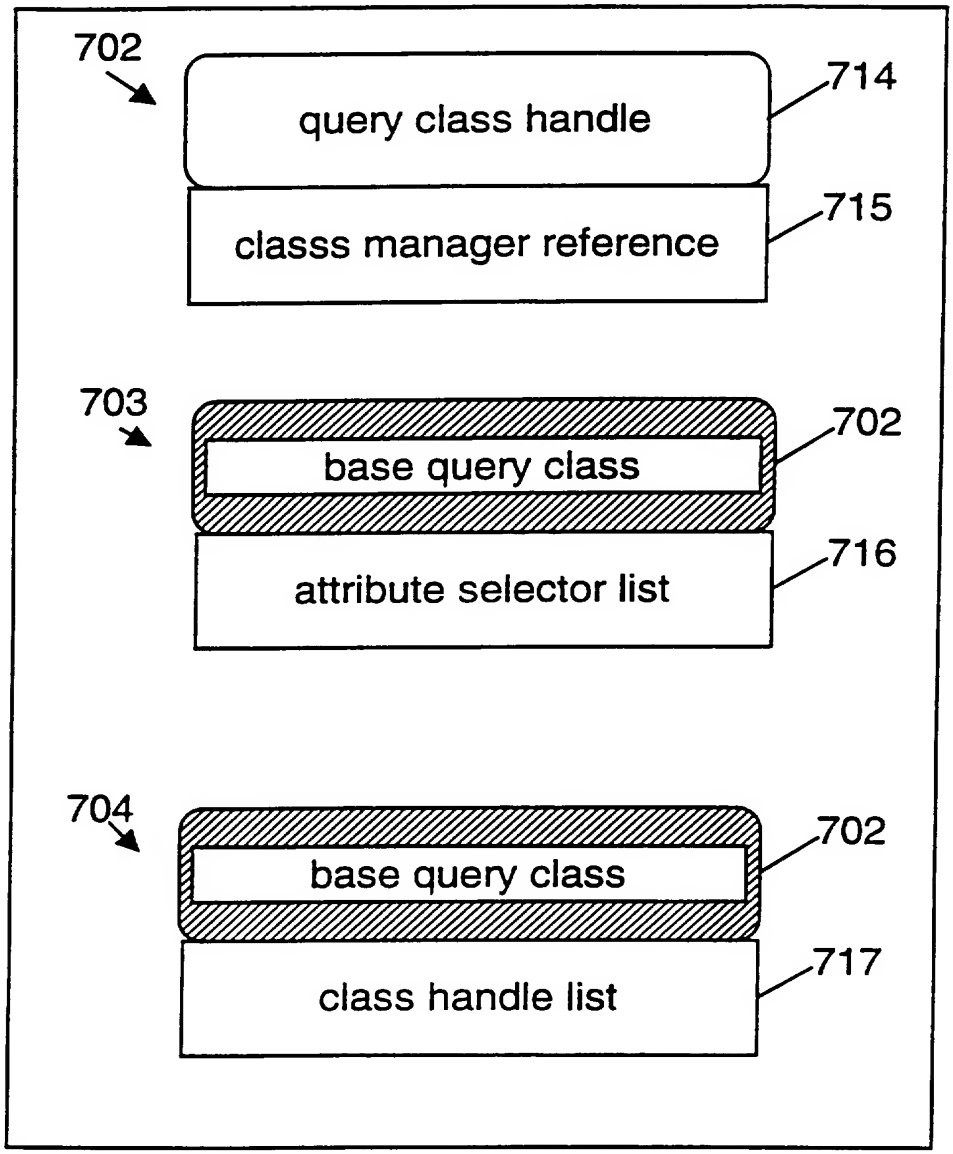

702
query class handle
714
classs manager reference
715
703
base query class
702
attribute selector list
716
704
base query class
702
class handle list
717

FIG. 162

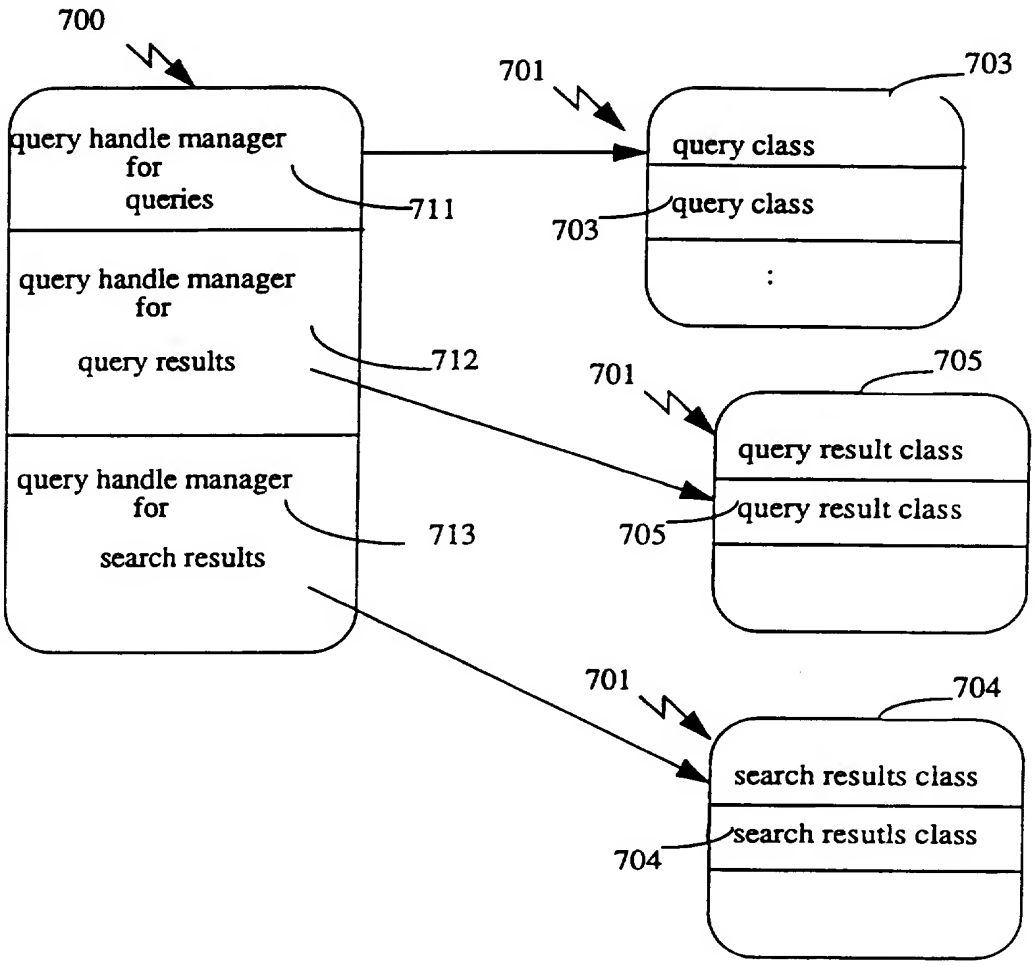

700
701
703
query handle manager for queries
711
query class
query class
703
:
query handle manager for query results
712
701
705
query result class
query result class
705
query handle manager for search results
713
701
704
search results class
search resutls class
704

FIG. 163

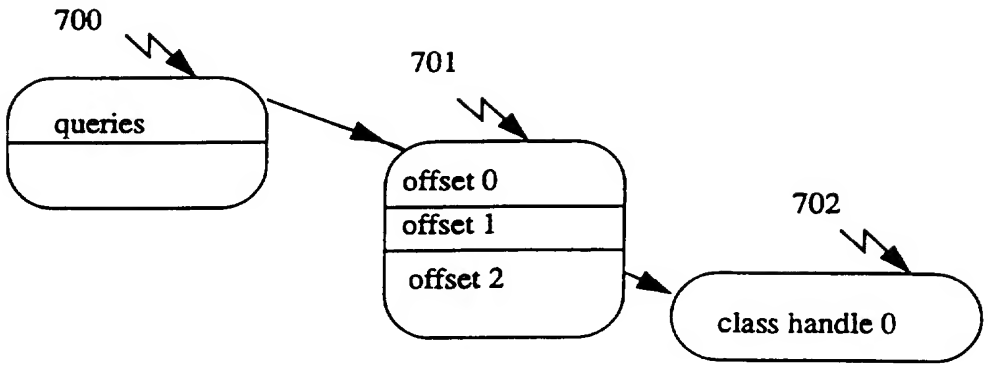


FIG. 164

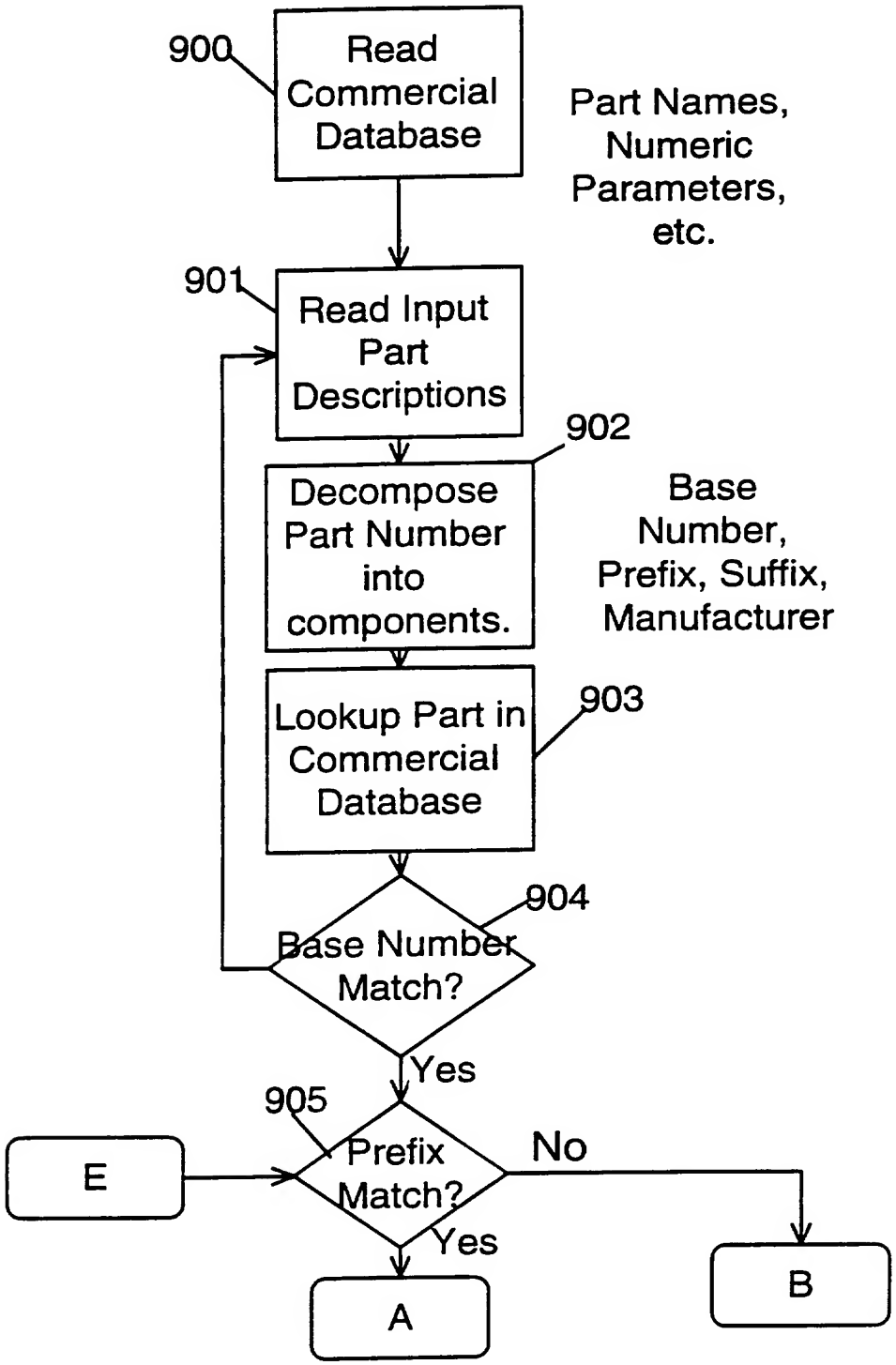

900
Read Commercial Database
Part Names, Numeric Parameters, etc.
901
Read Input Part Descriptions
902
Decompose Part Number into components.
Base Number, Prefix, Suffix, Manufacturer
903
Lookup Part in Commercial Database
904
Base Number Match?
Yes
905
Prefix Match?
No
Yes
E
A
B

FIG. 165

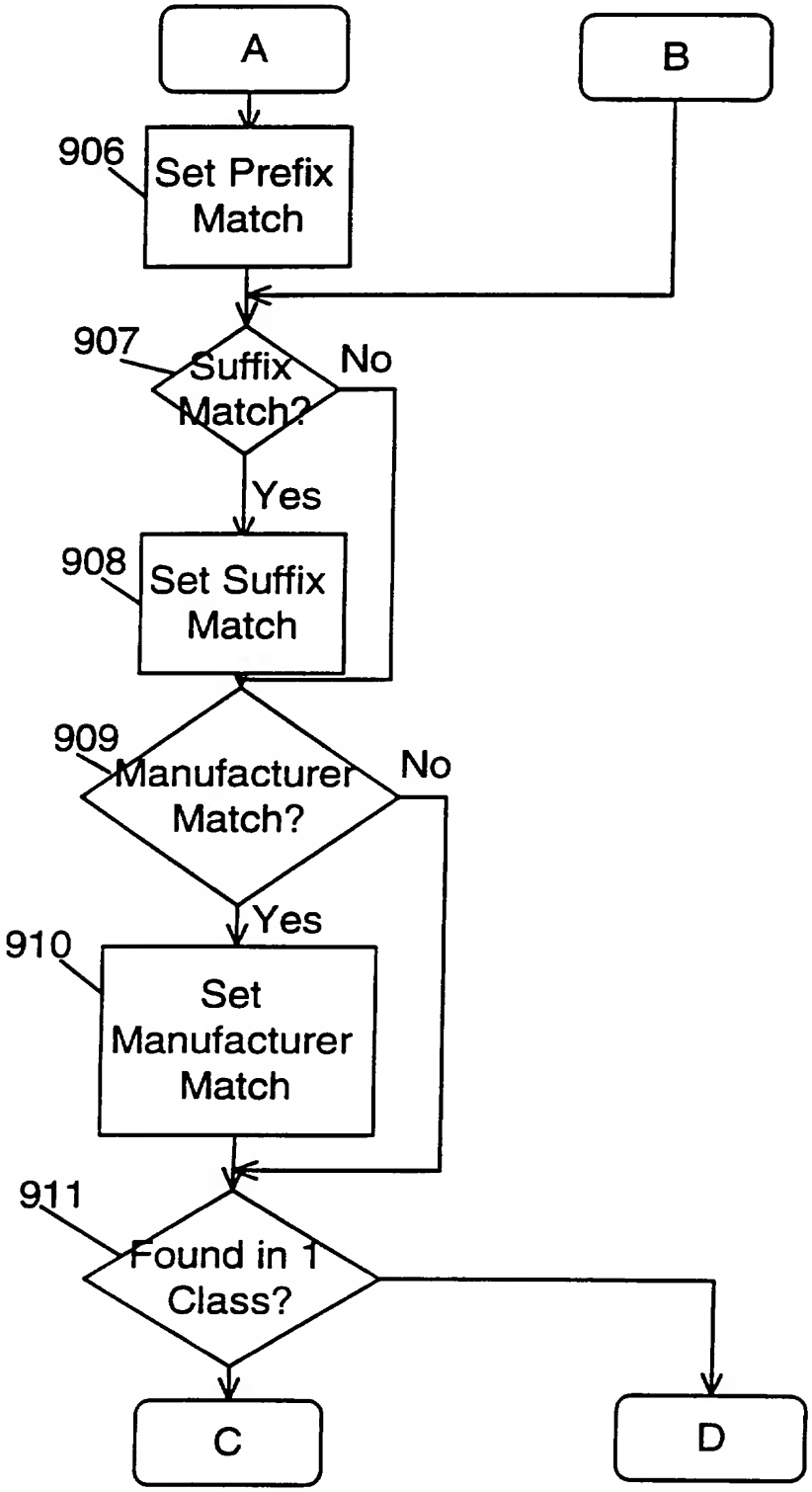

A
B
906
Set Prefix Match
907
Suffix Match?
No
Yes
908
Set Suffix Match
909
Manufacturer Match?
No
Yes
910
Set Manufacturer Match
911
Found in 1 Class?
C
D

FIG. 166

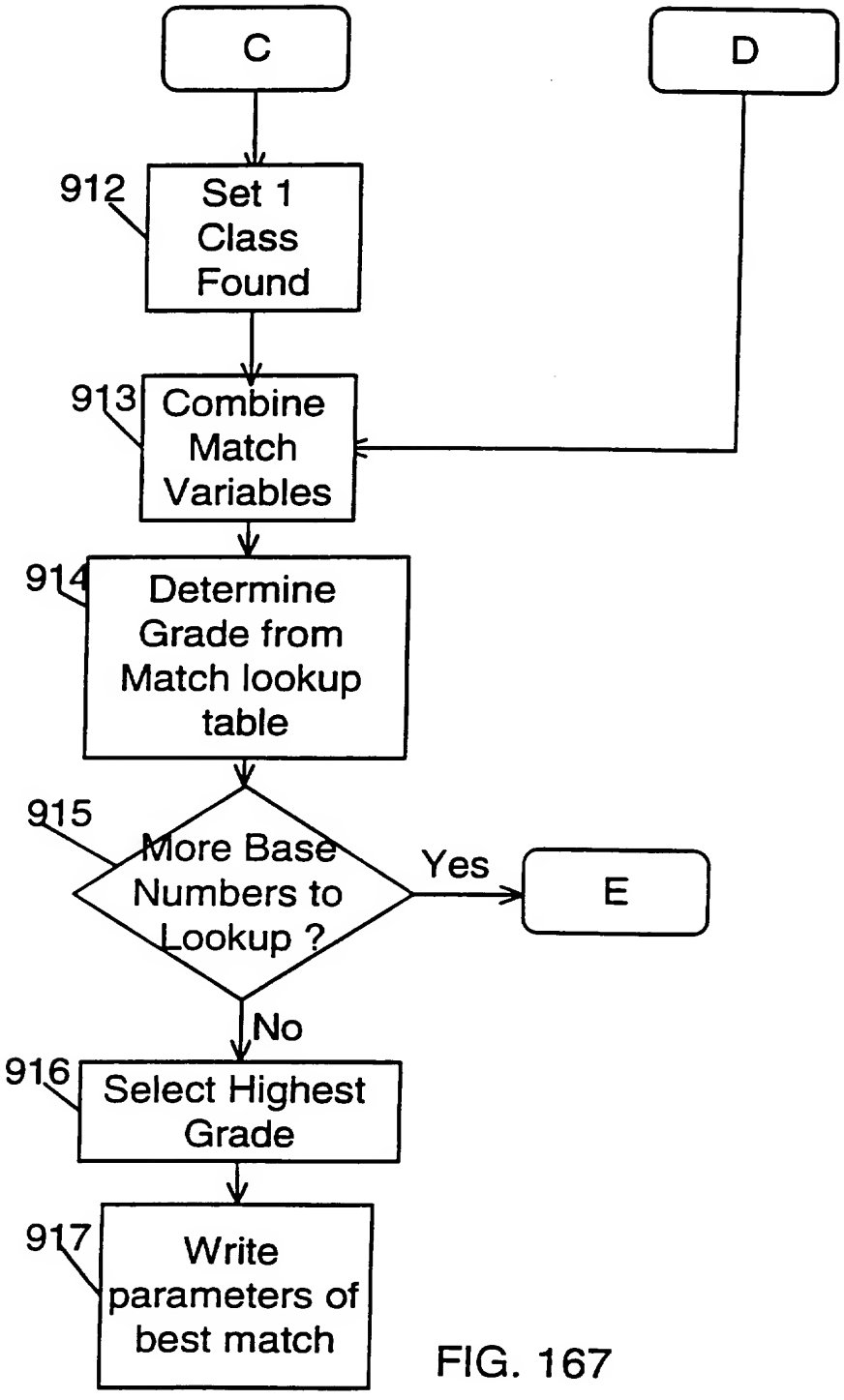

C
D
912
Set 1 Class Found
913
Combine Match Variables
914
Determine Grade from Match lookup table
915
More Base Numbers to Lookup ?
Yes
E
No
916
Select Highest Grade
917
Write parameters of best match

FIG. 167

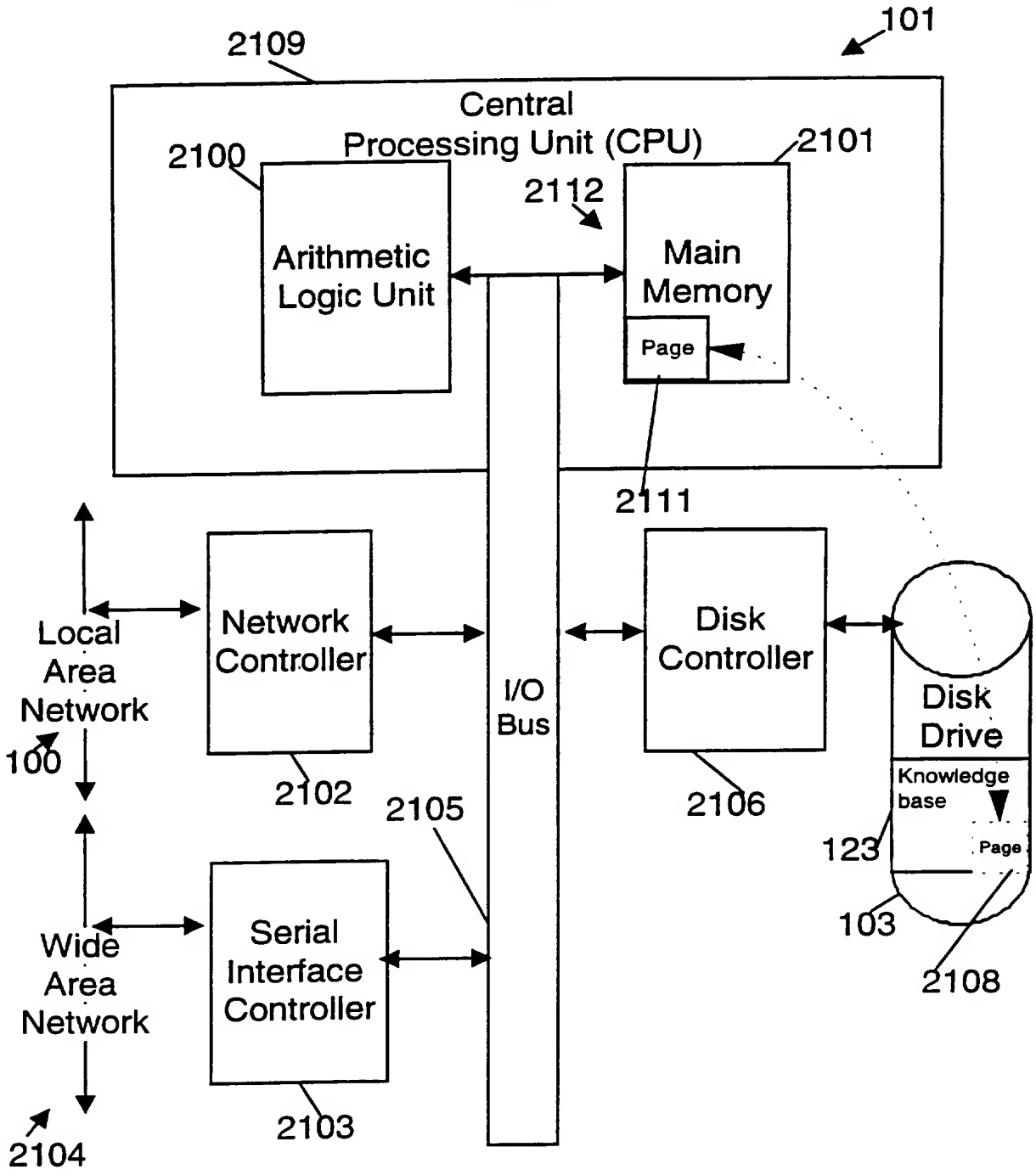

101
2109
Central
Processing Unit (CPU)
2100
2101
2112
Arithmetic
Logic Unit
Main
Memory
Page
2111
Local
Area
Network
100
Network
Controller
2102
I/O
Bus
Disk
Controller
2106
Disk
Drive
Knowledge
base
Page
123
103
2108
2105
Wide
Area
Network
Serial
Interface
Controller
2103
2104

FIG. 168

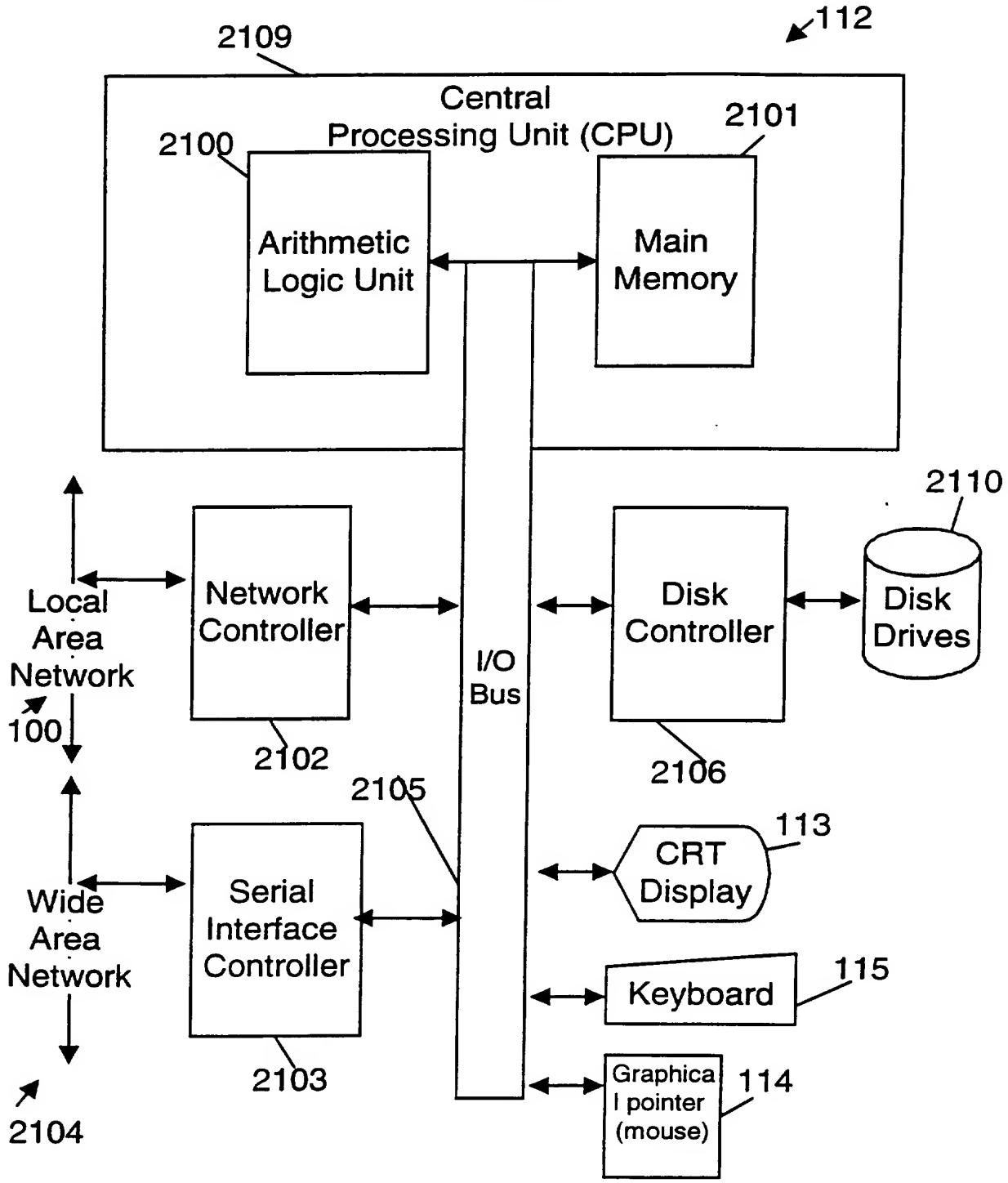

112
2109
Central
Processing Unit (CPU)
2101
2100
Arithmetic
Logic Unit
Main
Memory
2110
Local
Area
Network
100
Network
Controller
I/O
Bus
Disk
Controller
Disk
Drives
2102
2106
2105
113
CRT
Display
Wide
Area
Network
Serial
Interface
Controller
Keyboard
115
2103
Graphica
l pointer
(mouse)
114
2104

FIG. 169

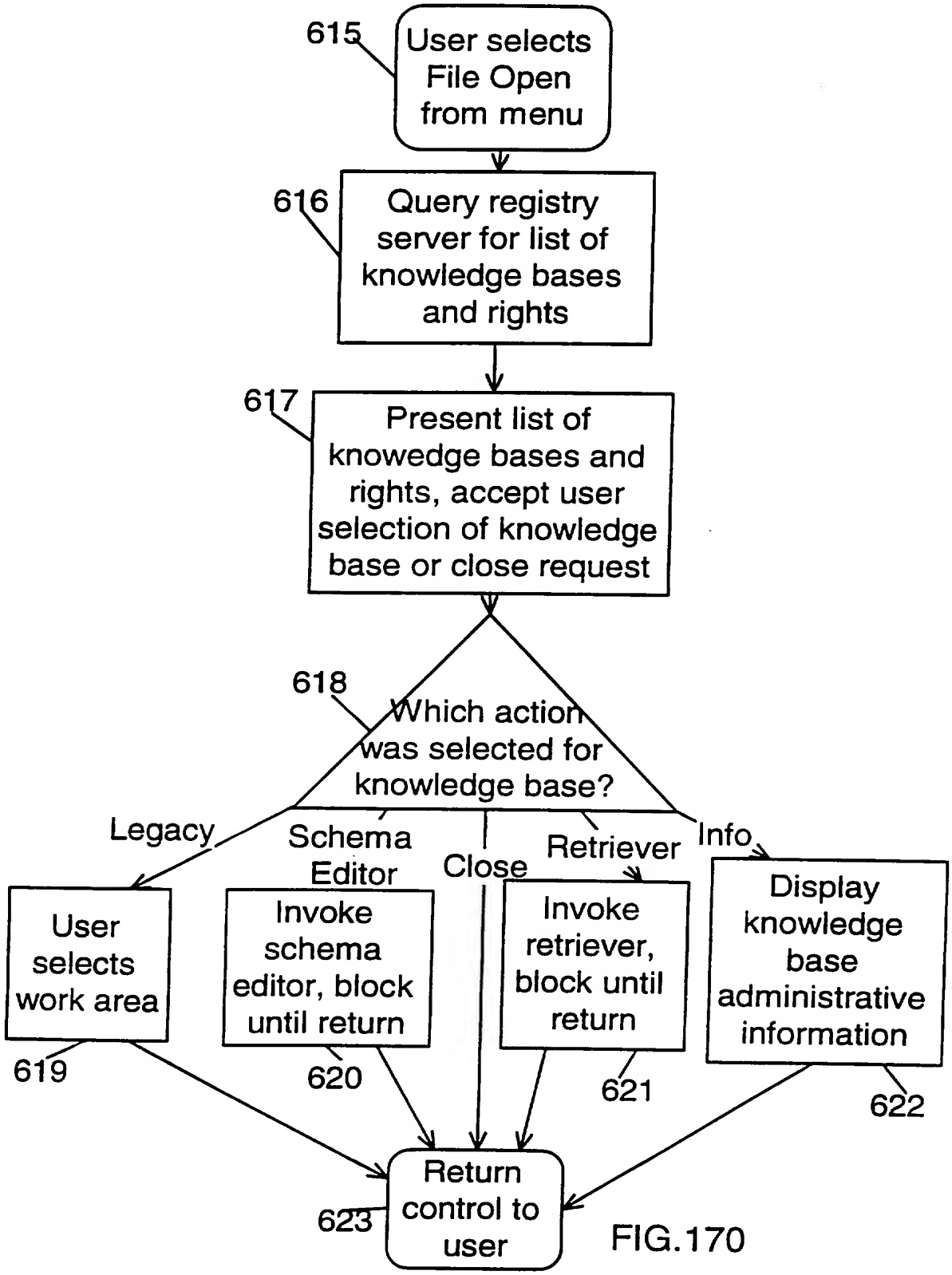

615
User selects File Open from menu
616
Query registry server for list of knowledge bases and rights
617
Present list of knowedge bases and rights, accept user selection of knowledge base or close request
618
Which action was selected for knowledge base?
Legacy
Schema Editor
Close
Retriever
Info
User selects work area
Invoke schema editor, block until return
Invoke retriever, block until return
Display knowledge base administrative information
619
620
621
622
Return control to user
623

FIG.170

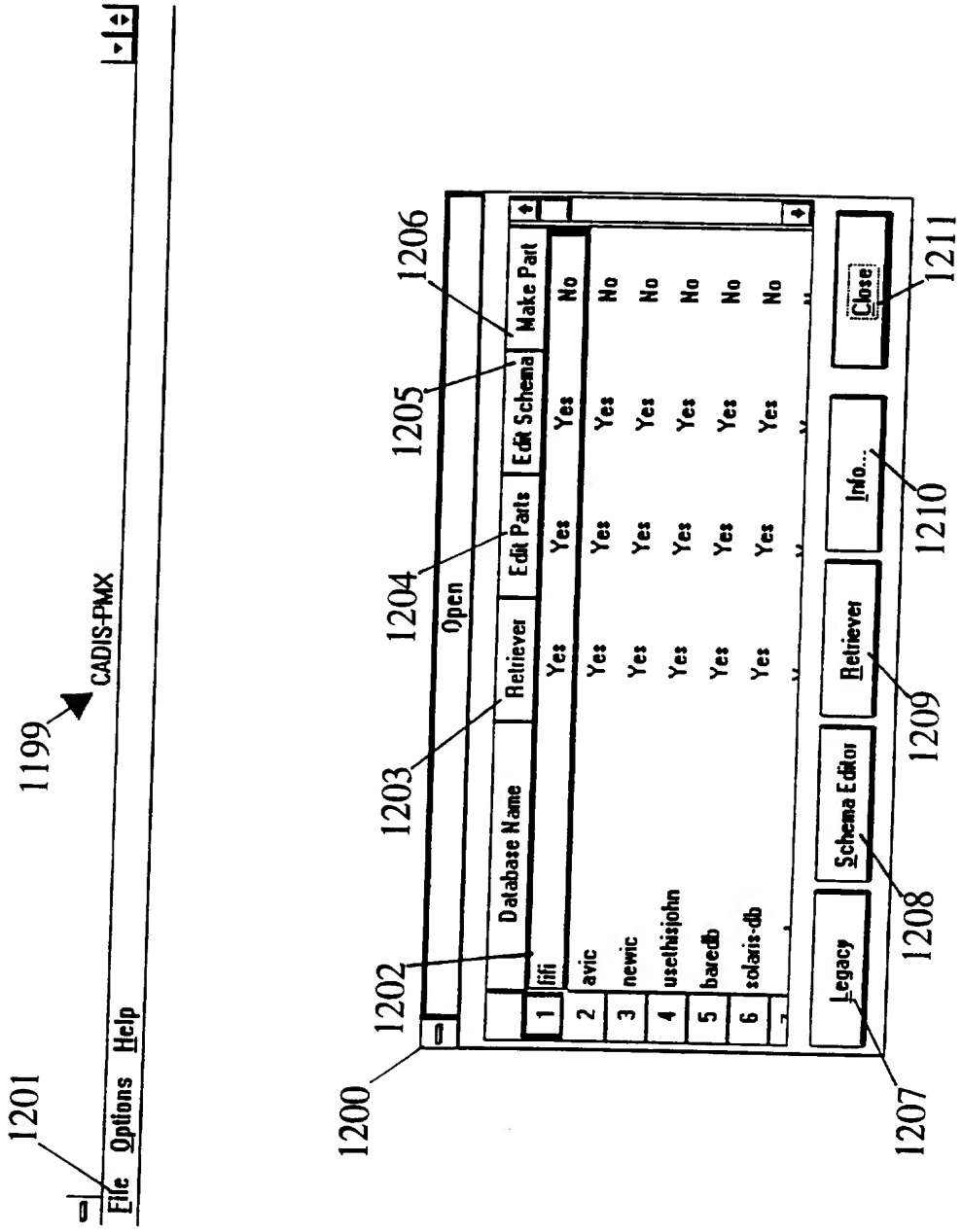

1199
CADIS-PMX
1201
File Options Help
1200
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
Open
Database Name
Retriever
Edit Parts
Edit Schema
Make Part
Close
Info...
Retriever
Schema Editor
Legacy
1 fifi Yes Yes Yes No
2 avic Yes Yes Yes No
3 newic Yes Yes Yes No
4 usethisjohn Yes Yes Yes No
5 baredb Yes Yes Yes No
6 solaris-db Yes Yes Yes No

FIG. 171

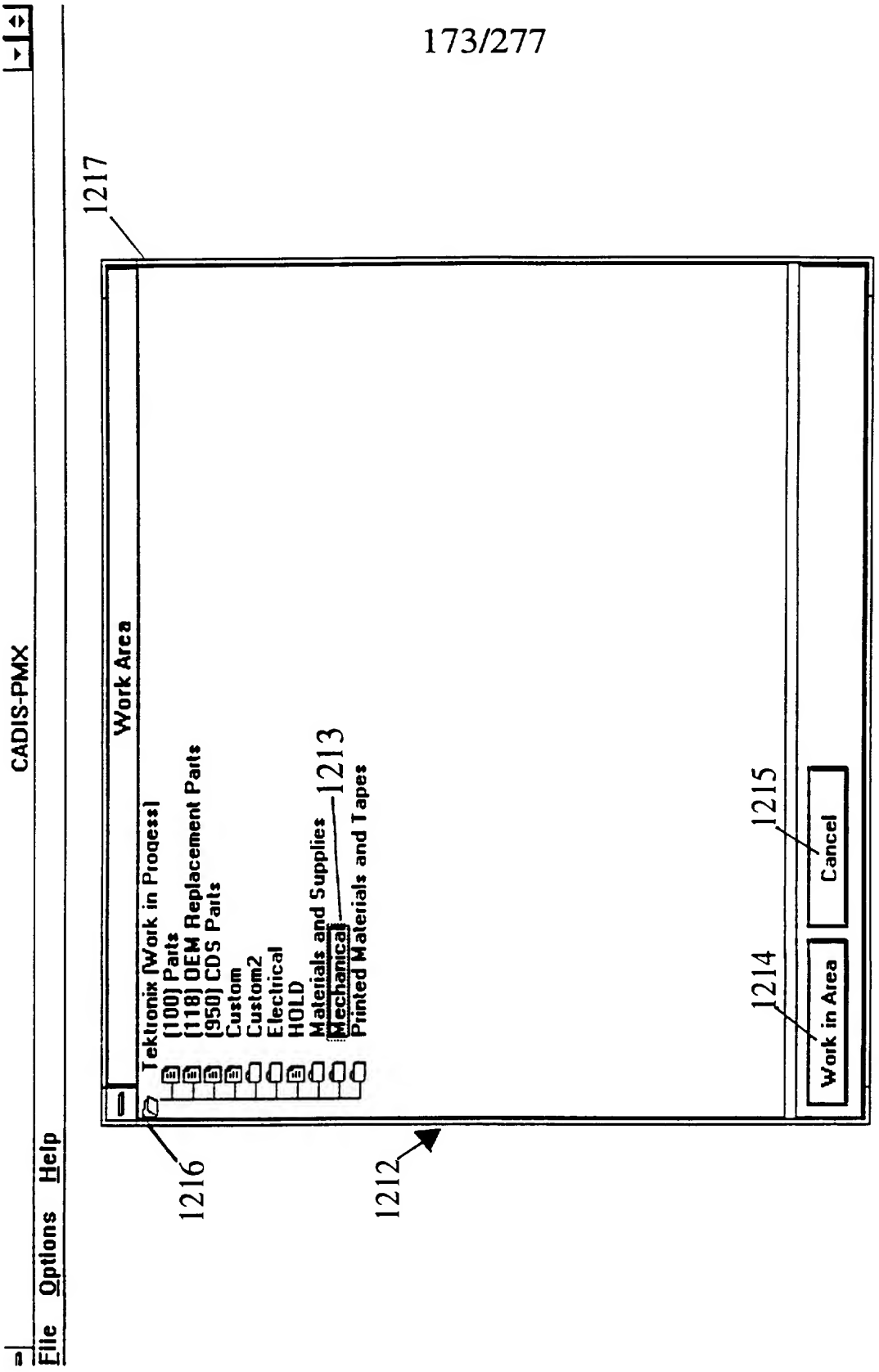

CADIS-PMX
File Options Help
Work Area
Tektronix (Work in Progess)
(100) Parts
(118) OEM Replacement Parts
(950) CDS Parts
Custom
Custom2
Electrical
HOLD
Materials and Supplies
Mechanical
Printed Materials and Tapes
Work in Area
Cancel
1212
1213
1214
1215
1216
1217

FIG. 172

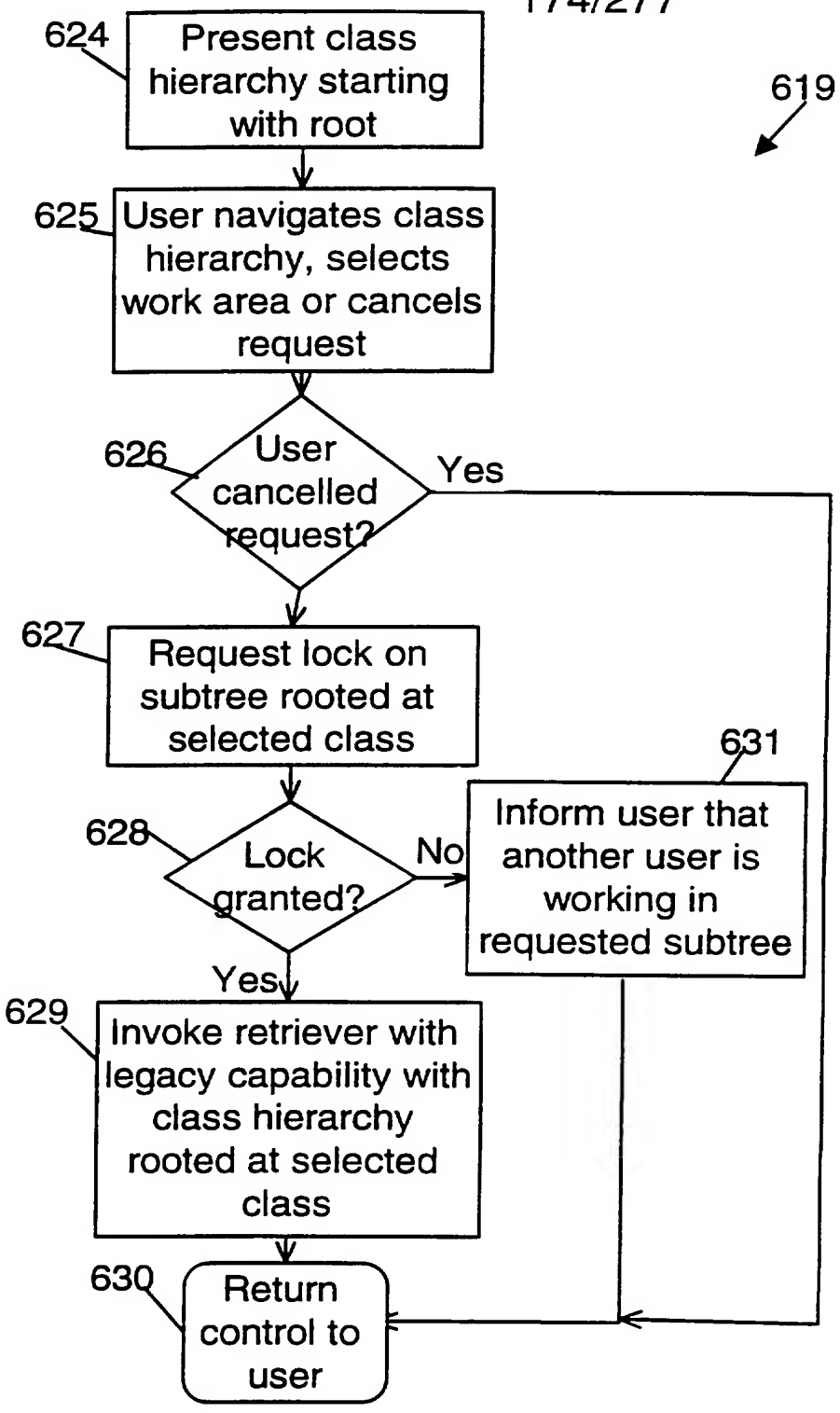

624
Present class hierarchy starting with root
619
625
User navigates class hierarchy, selects work area or cancels request
626
User cancelled request?
Yes
627
Request lock on subtree rooted at selected class
631
628
Lock granted?
No
Inform user that another user is working in requested subtree
Yes
629
Invoke retriever with legacy capability with class hierarchy rooted at selected class
630
Return control to user

FIG.173

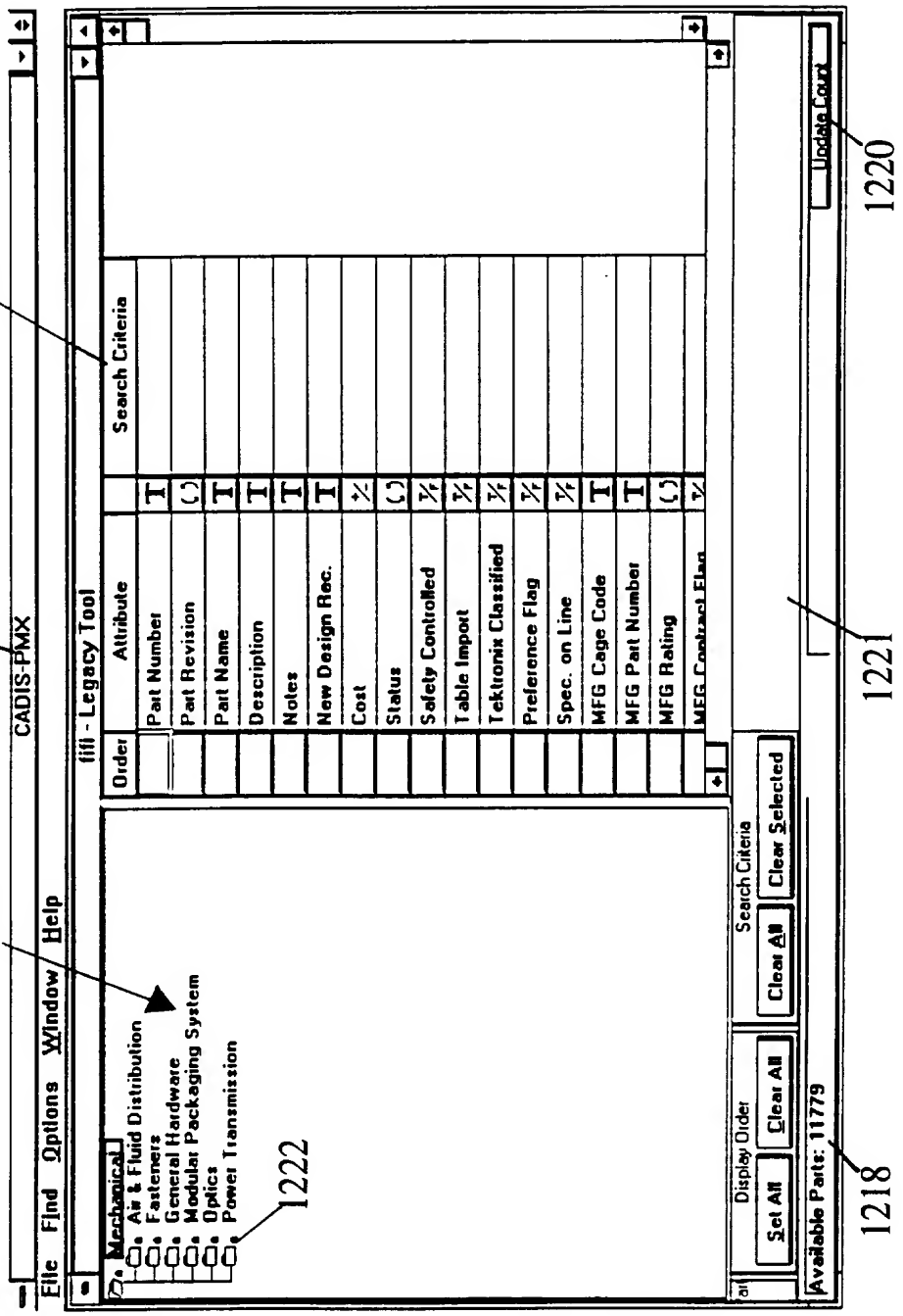

1224
1219
1220
1223
1221
1222
1218
CADIS-PMX
File Find Options Window Help
fifi - Legacy Tool
Mechanical
Air & Fluid Distribution
Fasteners
General Hardware
Modular Packaging System
Optics
Power Transmission
Order
Attribute
Search Criteria
Part Number
Part Revision
Part Name
Description
Notes
New Design Rec.
Cost
Status
Safety Controlled
Table Import
Tektronix Classified
Preference Flag
Spec. on Line
MFG Cage Code
MFG Part Number
MFG Rating
Update Count
Search Criteria
Display Order
Set All
Clear All
Clear All
Clear Selected
Available Parts: 11779

FIG. 174

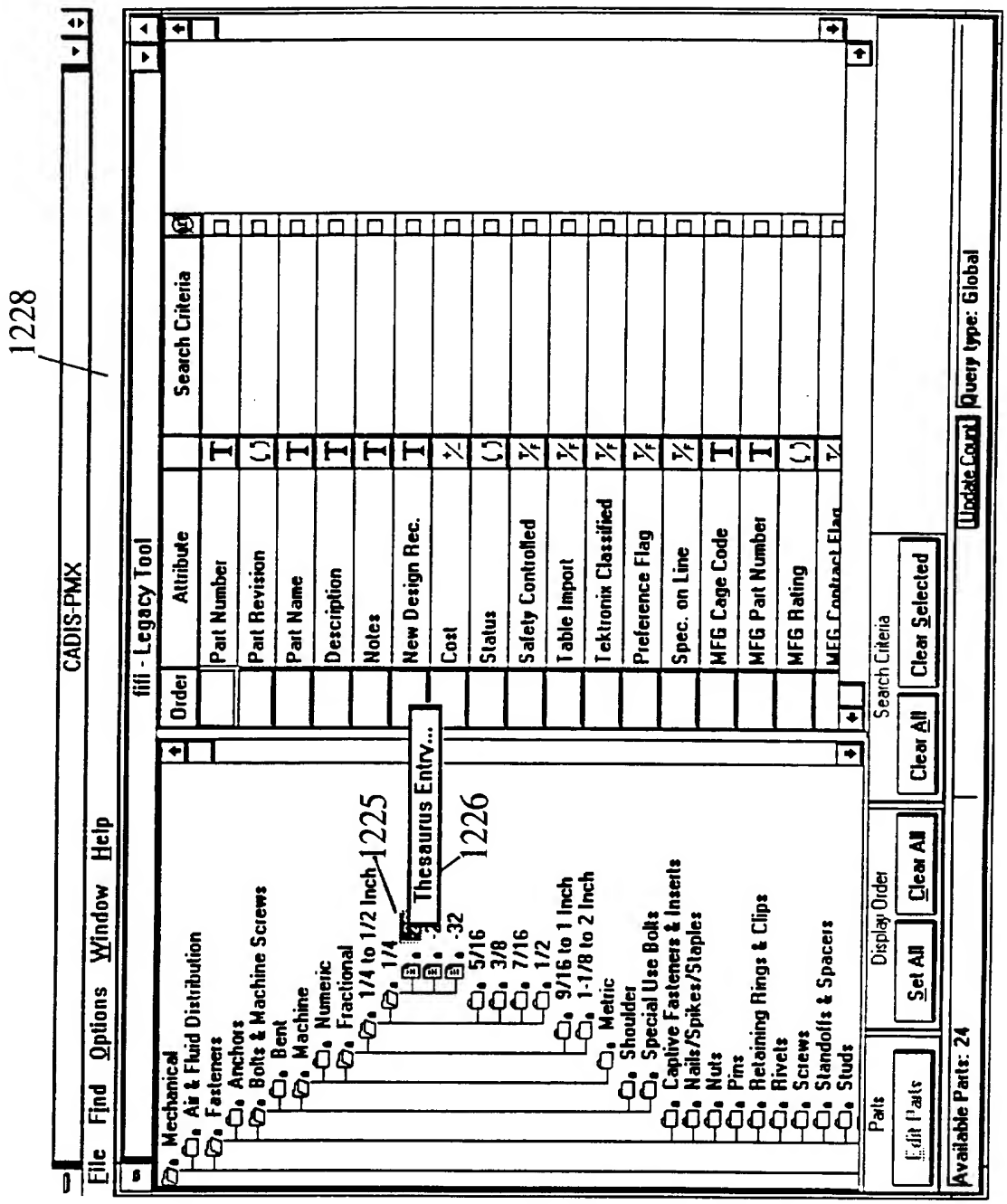

1228
CADIS-PMX
File Find Options Window Help
fifi - Legacy Tool
Mechanical
Air & Fluid Distribution
Fasteners
Anchors
Bolts & Machine Screws
Bent
Machine
Numeric
Fractional
1/4 to 1/2 Inch
1/4
-32
5/16
3/8
7/16
1/2
9/16 to 1 Inch
1-1/8 to 2 Inch
Metric
Shoulder
Special Use Bolts
Captive Fasteners & Inserts
Nails/Spikes/Staples
Nuts
Pins
Retaining Rings & Clips
Rivets
Screws
Standoffs & Spacers
Studs
1225
Thesaurus Entry...
1226
Order
Attribute
Search Criteria
Part Number
Part Revision
Part Name
Description
Notes
New Design Rec.
Cost
Status
Safety Controlled
Table Import
Tektronix Classified
Preference Flag
Spec. on Line
MFG Cage Code
MFG Part Number
MFG Rating
Parts
Edit Parts
Display Order
Set All
Clear All
Search Criteria
Clear All
Clear Selected
Update Count
Query type: Global
Available Parts: 24

FIG. 175

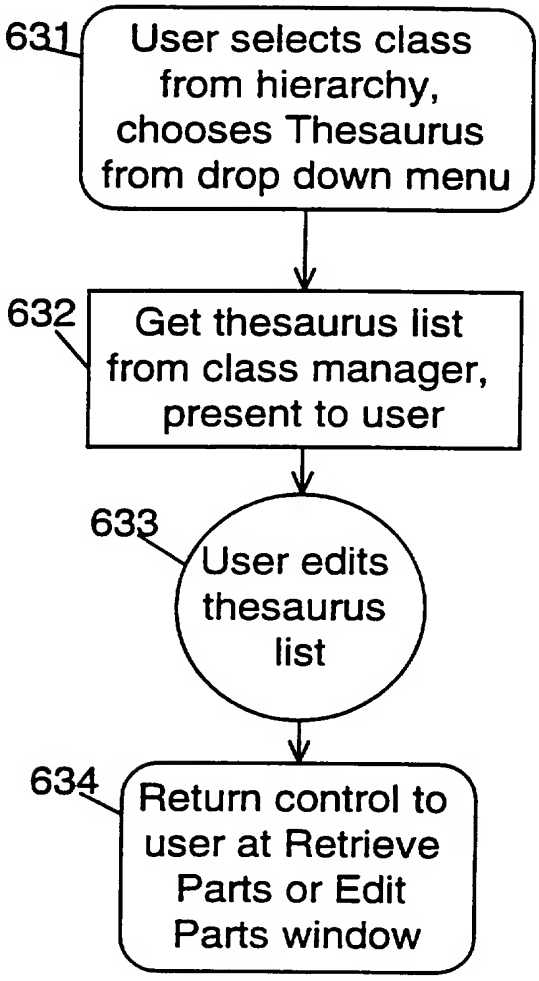

631
User selects class from hierarchy, chooses Thesaurus from drop down menu
632
Get thesaurus list from class manager, present to user
633
User edits thesaurus list
634
Return control to user at Retrieve Parts or Edit Parts window

FIG.176

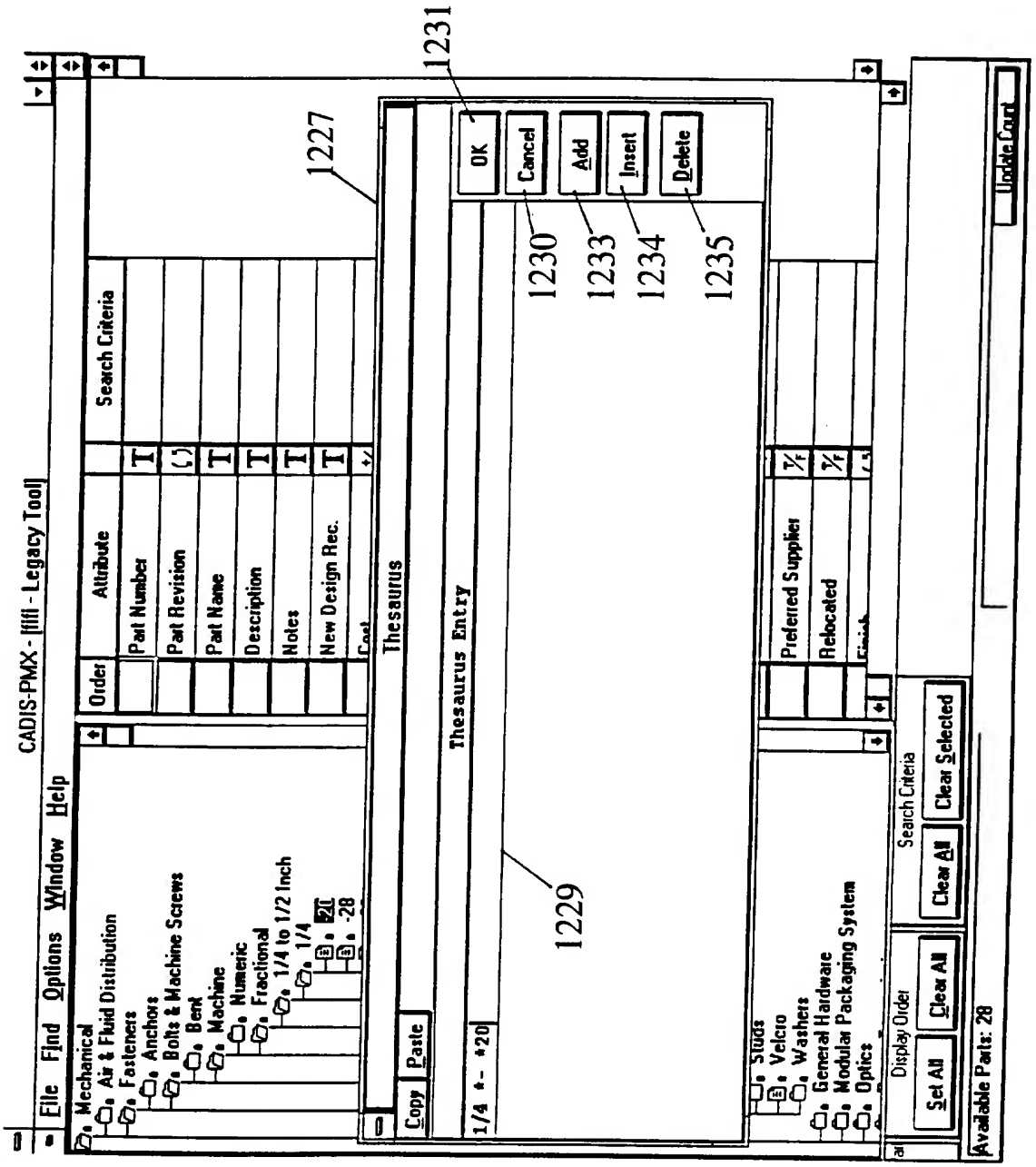

1231
1227
1230
1233
1234
1235
1229
CADIS-PMX - [fifi - Legacy Tool]
File Find Options Window Help
Mechanical
Air & Fluid Distribution
Fasteners
Anchors
Bolts & Machine Screws
Bent
Machine
Numeric
Fractional
1/4 to 1/2 Inch
1/4
#20
#28
Order
Attribute
Search Criteria
Part Number
Part Revision
Part Name
Description
Notes
New Design Rec.
Cost
Thesaurus
Copy
Paste
Thesaurus Entry
1/4 +- #20
OK
Cancel
Add
Insert
Delete
Preferred Supplier
Relocated
Studs
Velcro
Washers
General Hardware
Modular Packaging System
Optics
Display Order
Set All
Clear All
Search Criteria
Clear All
Clear Selected
Update Count
Available Parts: 28

FIG. 177

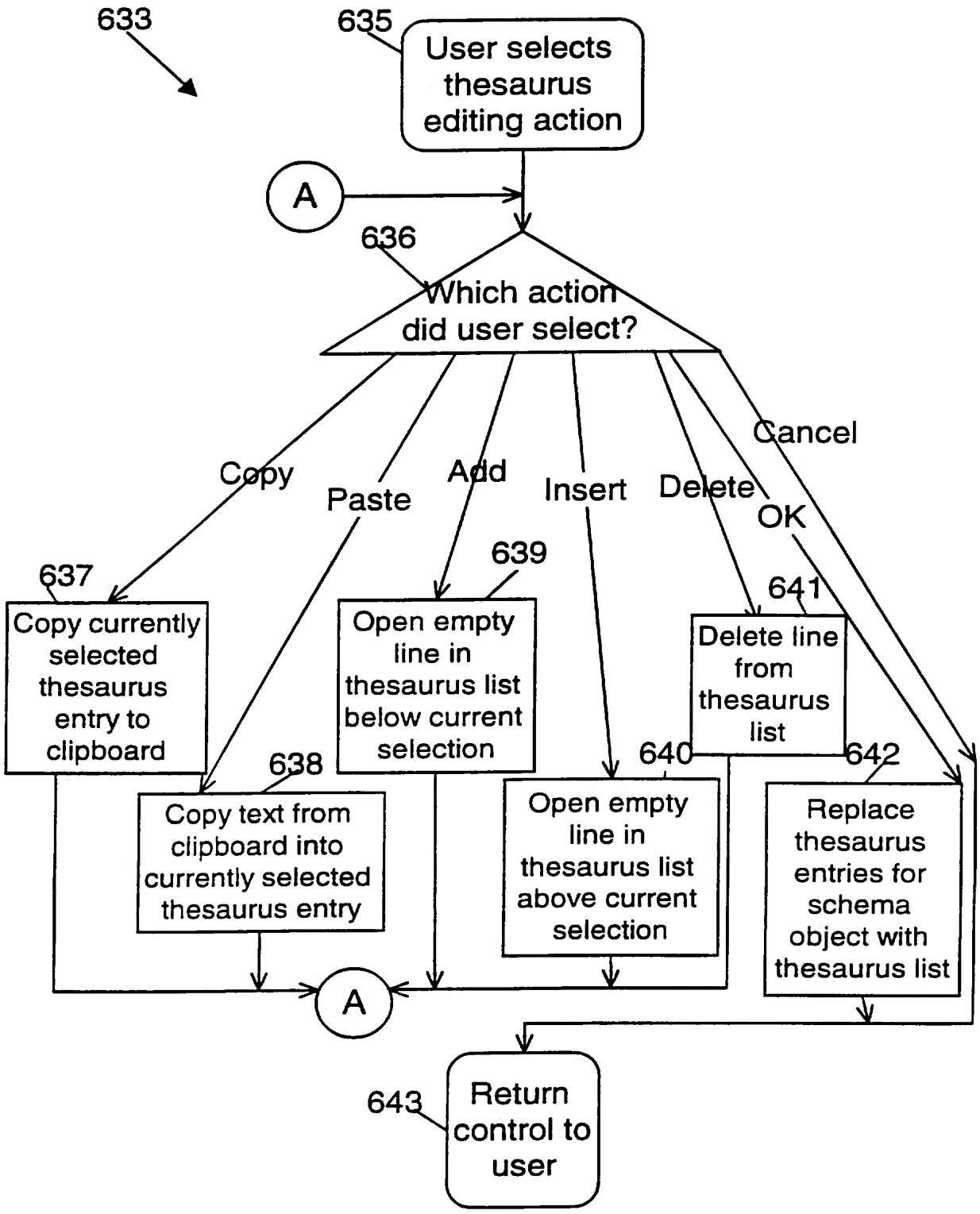

633
635
User selects thesaurus editing action
A
636
Which action did user select?
Copy
Paste
Add
Insert
Delete
OK
Cancel
637
Copy currently selected thesaurus entry to clipboard
639
Open empty line in thesaurus list below current selection
641
Delete line from thesaurus list
638
Copy text from clipboard into currently selected thesaurus entry
640
Open empty line in thesaurus list above current selection
642
Replace thesaurus entries for schema object with thesaurus list
A
643
Return control to user

FIG.178

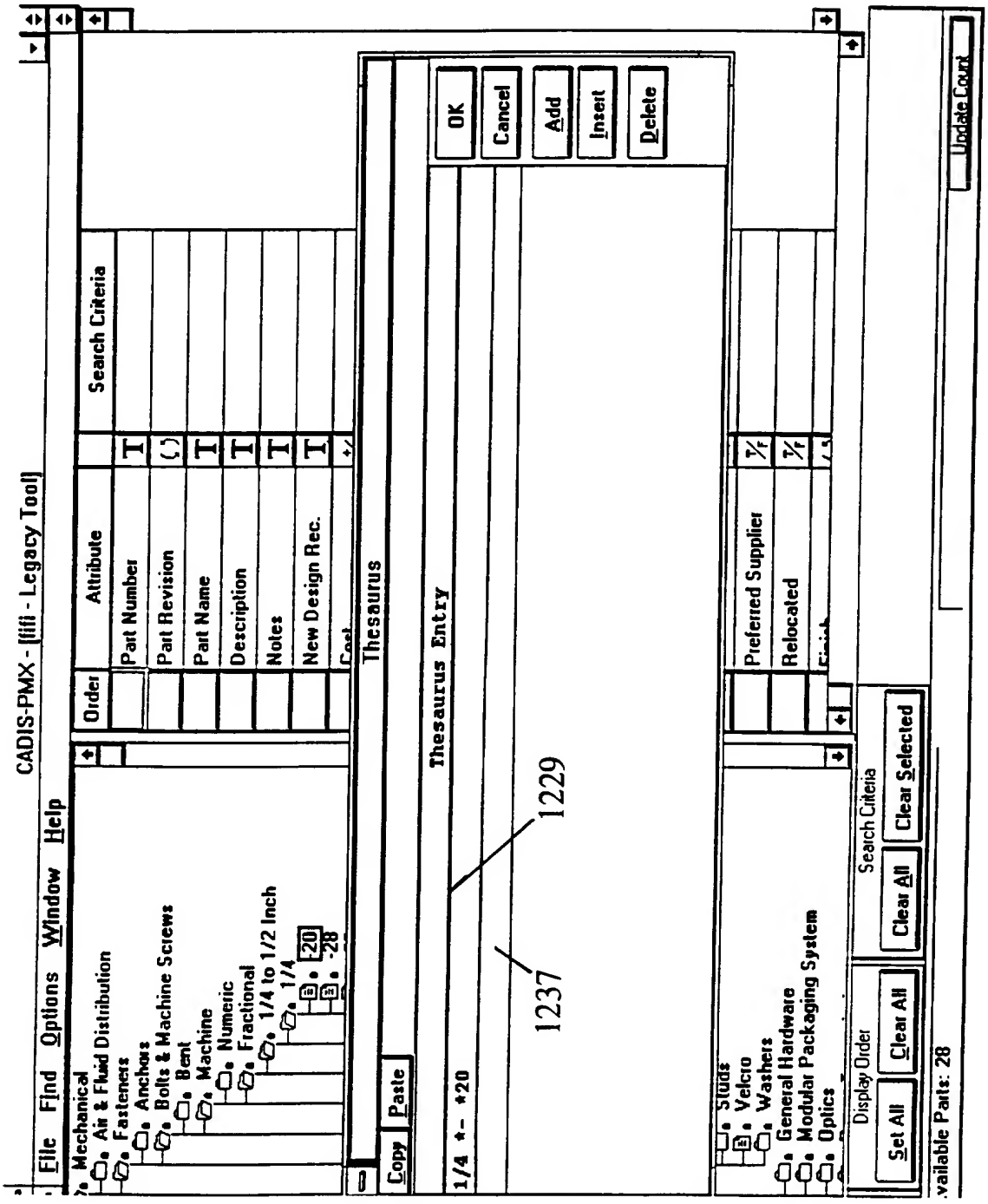

CADIS-PMX - [fifi - Legacy Tool]
File Find Options Window Help
Mechanical
Air & Fluid Distribution
Fasteners
Anchors
Bolts & Machine Screws
Bent
Machine
Numeric
Fractional
1/4 to 1/2 Inch
1/4
20
28
Order
Attribute
Search Criteria
Part Number
Part Revision
Part Name
Description
Notes
New Design Rec.
Cost
Thesaurus
Copy
Paste
Thesaurus Entry
1/4 +- +20
OK
Cancel
Add
Insert
Delete
1229
1237
Preferred Supplier
Relocated
Studs
Velcro
Washers
General Hardware
Modular Packaging System
Optics
Display Order
Set All
Clear All
Search Criteria
Clear All
Clear Selected
Update Count
vailable Parts: 28

FIG. 179

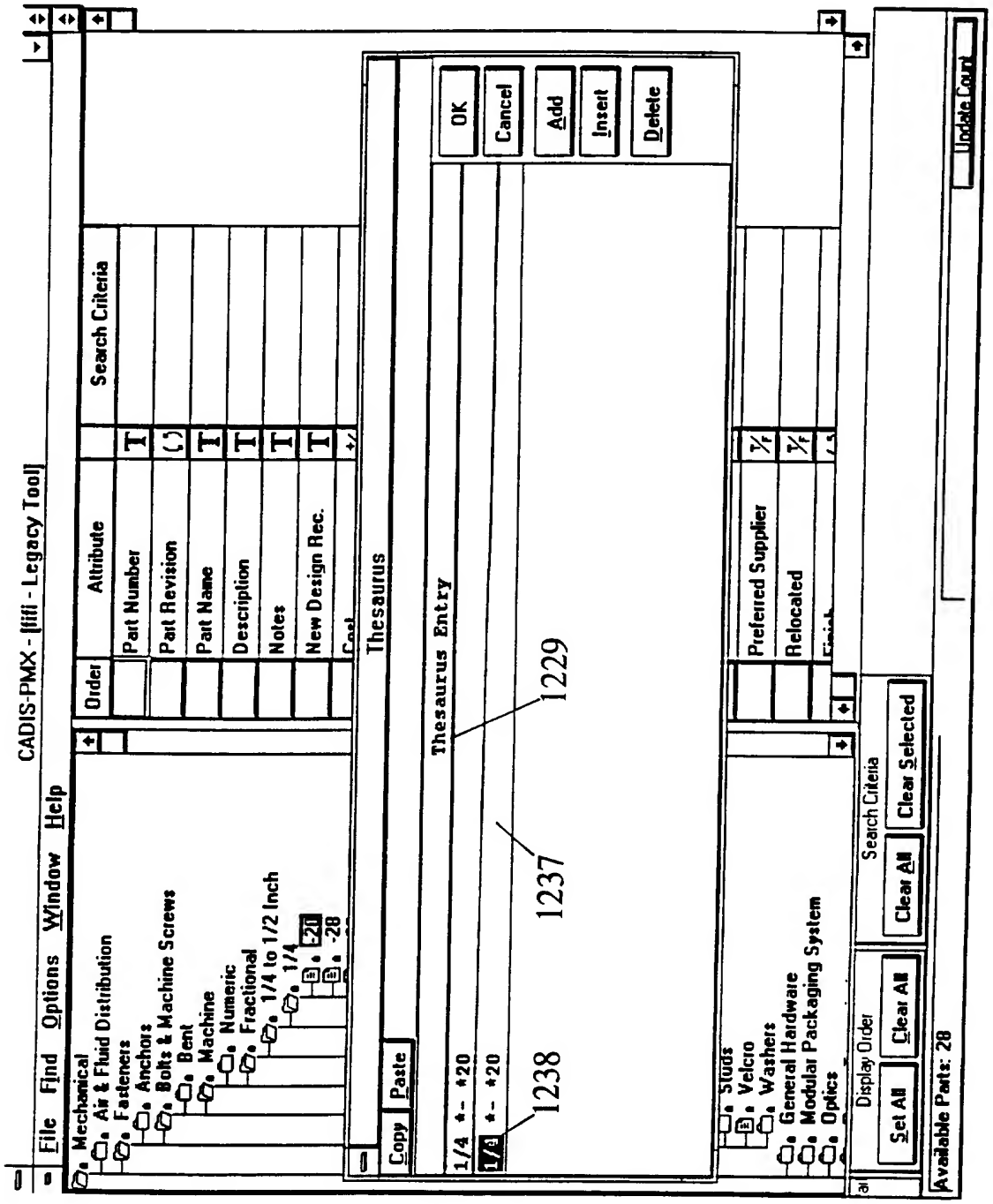

CADIS-PMX - [fill - Legacy Tool]
File Find Options Window Help
Mechanical
Air & Fluid Distribution
Fasteners
Anchors
Bolts & Machine Screws
Bent
Machine
Numeric
Fractional
1/4 to 1/2 Inch
1/4
Order
Attribute
Search Criteria
Part Number
Part Revision
Part Name
Description
Notes
New Design Rec.
Thesaurus
Copy
Paste
Thesaurus Entry
1/4 +- *20
1/4 +- *20
OK
Cancel
Add
Insert
Delete
1229
1237
1238
Preferred Supplier
Relocated
Studs
Velcro
Washers
General Hardware
Modular Packaging System
Optics
Display Order
Set All
Clear All
Search Criteria
Clear All
Clear Selected
Update Count
Available Parts: 28

FIG. 180

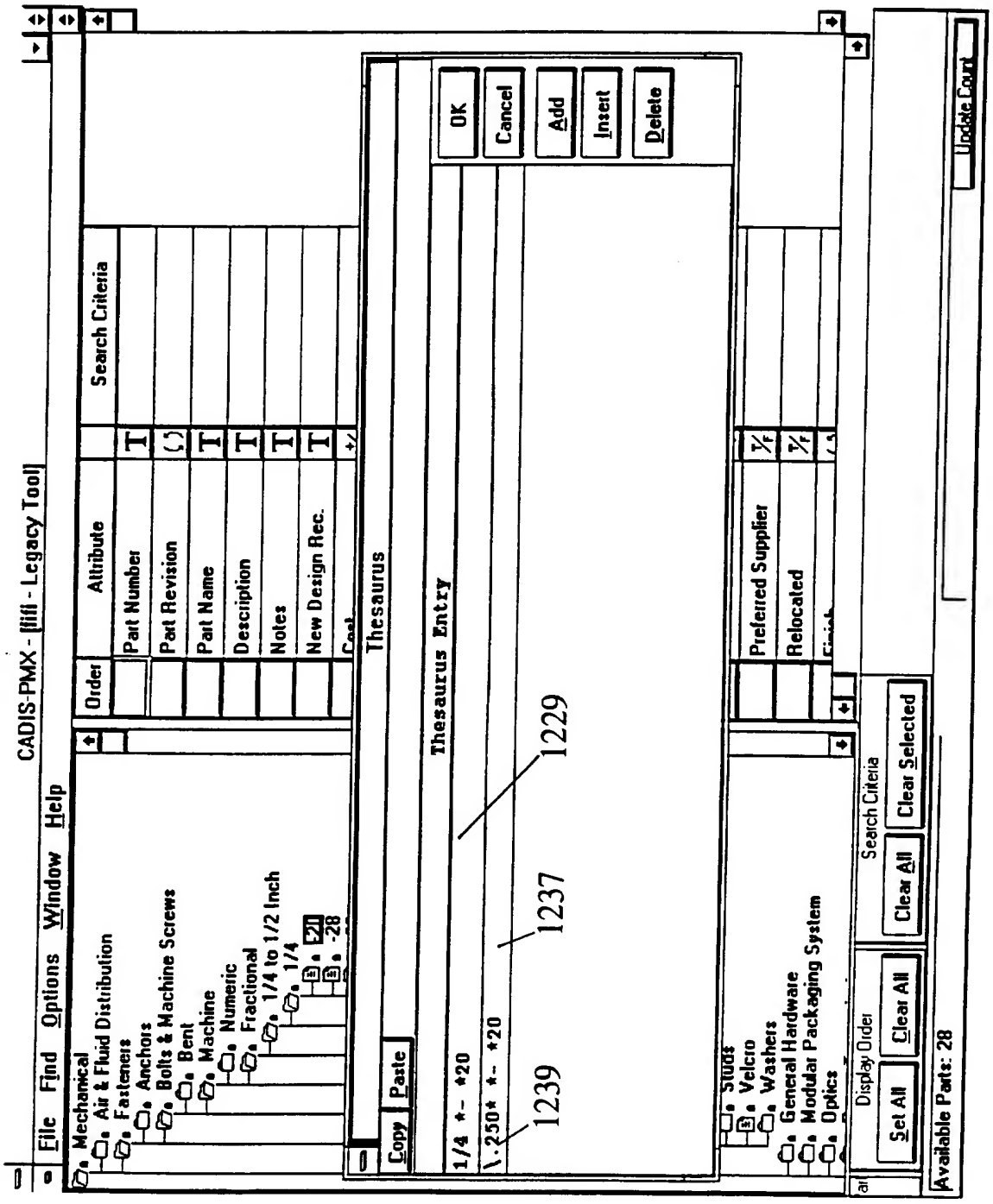

CADIS-PMX - [Hifi - Legacy Tool]
File Find Options Window Help
Mechanical
Air & Fluid Distribution
Fasteners
Anchors
Bolts & Machine Screws
Bent
Machine
Numeric
Fractional
1/4
1/4 to 1/2 Inch
Order
Attribute
Search Criteria
Part Number
Part Revision
Part Name
Desciption
Notes
New Design Rec.
Thesaurus
Copy
Paste
Thesaurus Entry
1/4 +- *20
\.250* +- *20
OK
Cancel
Add
Insert
Delete
1239
1237
1229
Preferred Supplier
Relocated
Studs
Velcro
Washers
General Hardware
Modular Packaging System
Optics
Display Order
Search Criteria
Set All
Clear All
Clear All
Clear Selected
Available Parts: 28
Update Count

FIG. 181

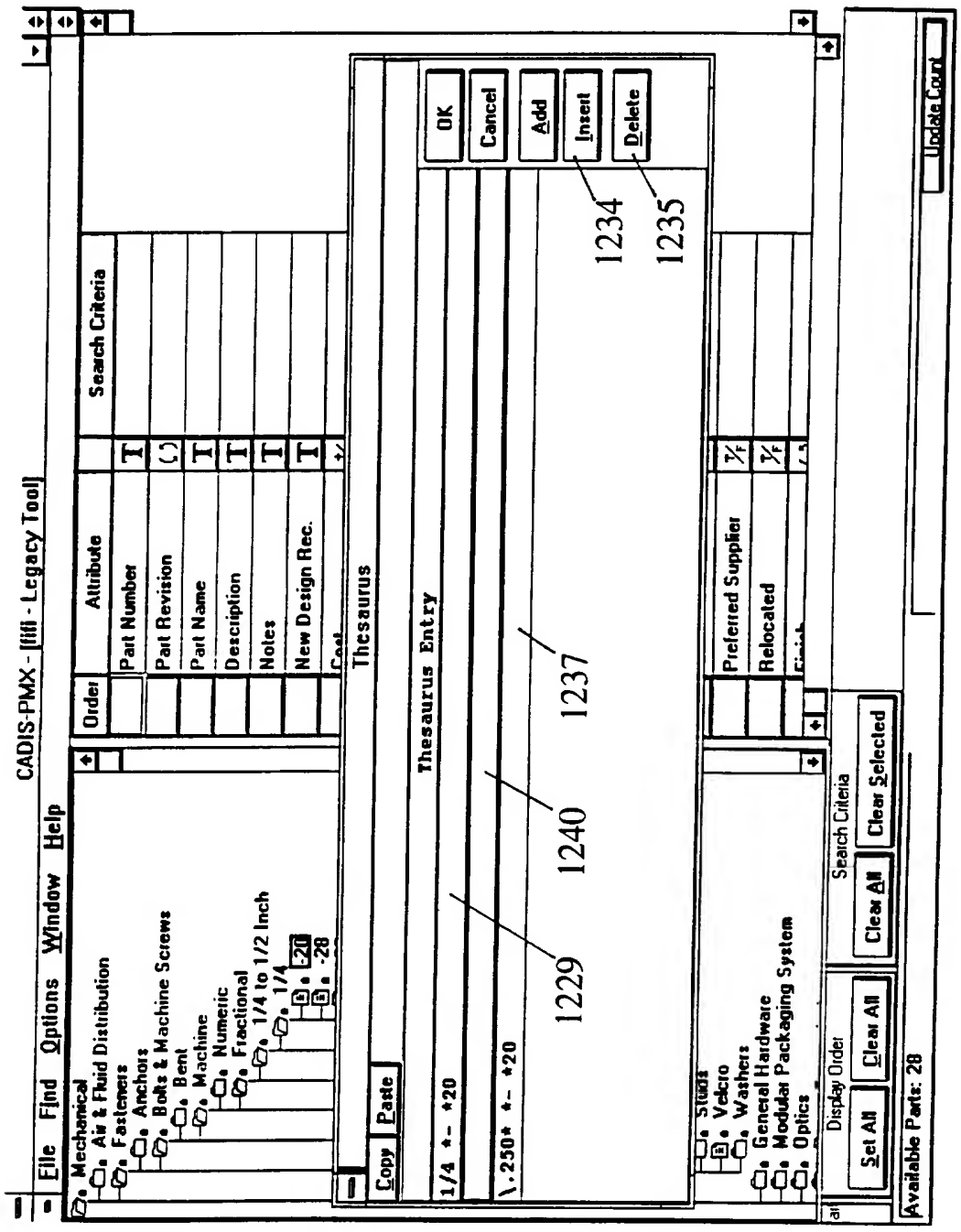

CADIS-PMX - [fifi - Legacy Tool]
File Find Options Window Help
Mechanical
Air & Fluid Distribution
Fasteners
Anchors
Bolts & Machine Screws
Bent
Machine
Numeric
Fractional
1/4 to 1/2 Inch
1/4
-20
-28
Order
Attribute
Search Criteria
Part Number
Part Revision
Part Name
Description
Notes
New Design Rec.
Cost
Thesaurus
Copy
Paste
Thesaurus Entry
1/4 +- .20
\.250* +- .20
OK
Cancel
Add
Insert
Delete
1234
1235
1229
1240
1237
Preferred Supplier
Relocated
Studs
Velcro
Washers
General Hardware
Modular Packaging System
Optics
Display Order
Set All
Clear All
Search Criteria
Clear All
Clear Selected
Update Count
Available Parts: 28

FIG. 182

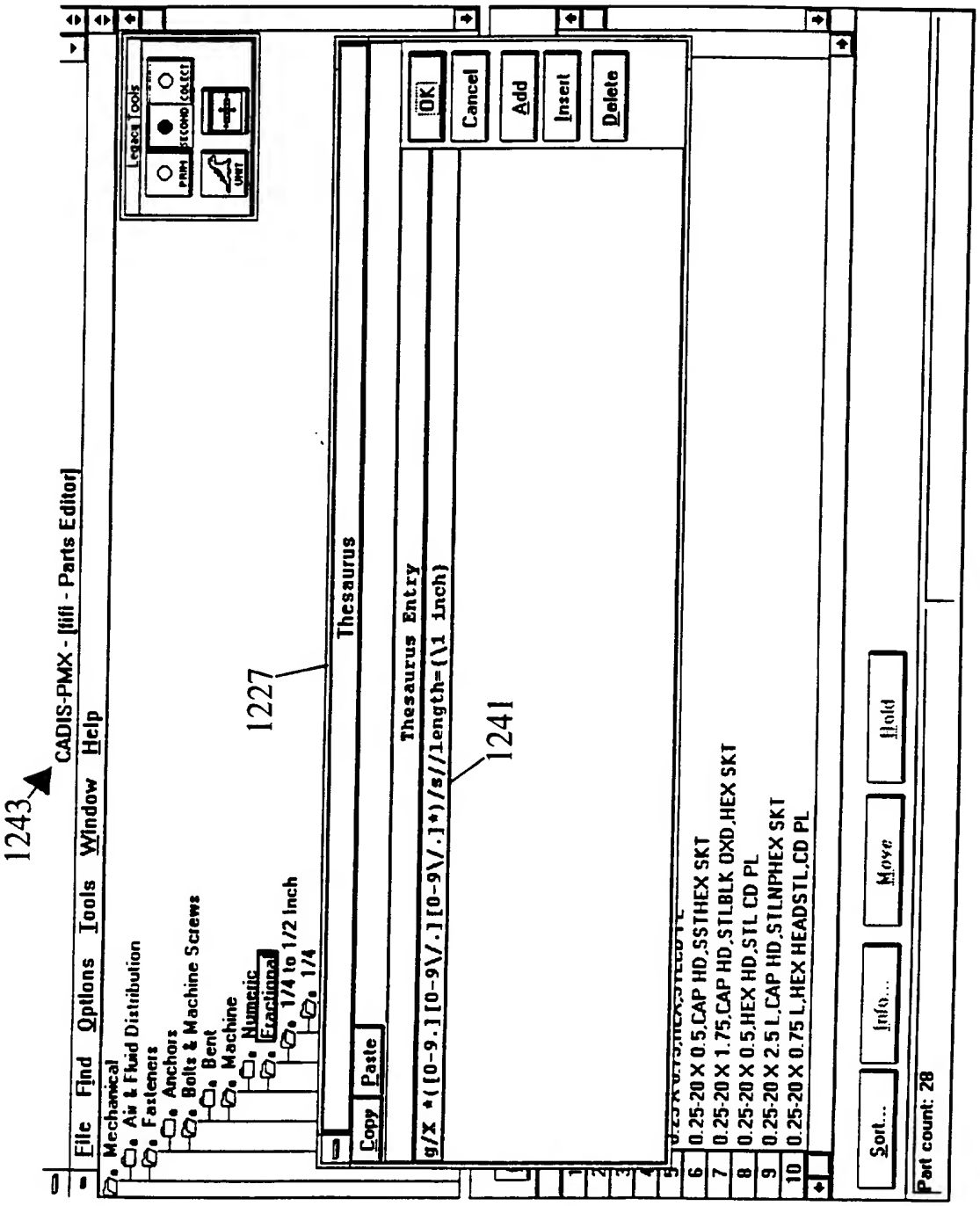

1243
CADIS-PMX - [fifi - Parts Editor]
File Find Options Tools Window Help
Mechanical
Air & Fluid Distribution
Fasteners
Anchors
Bolts & Machine Screws
Bent
Machine
Numeric
Fractional
1/4 to 1/2 Inch
1/4
Legacy Tools
PARM
SECOND
UNIT
1227
Thesaurus
Copy
Paste
Thesaurus Entry
g/X *([0-9.][0-9\/.]*)/s//length=(\1 inch)
1241
OK
Cancel
Add
Insert
Delete
6 0.25-20 X 0.5,CAP HD,SSTHEX SKT
7 0.25-20 X 1.75,CAP HD,STLBLK OXD,HEX SKT
8 0.25-20 X 0.5,HEX HD,STL CD PL
9 0.25-20 X 2.5 L,CAP HD,STLNPHEX SKT
10 0.25-20 X 0.75 L,HEX HEADSTL,CD PL
Sort...
Info...
Move
Hold
Part count: 28

FIG. 183

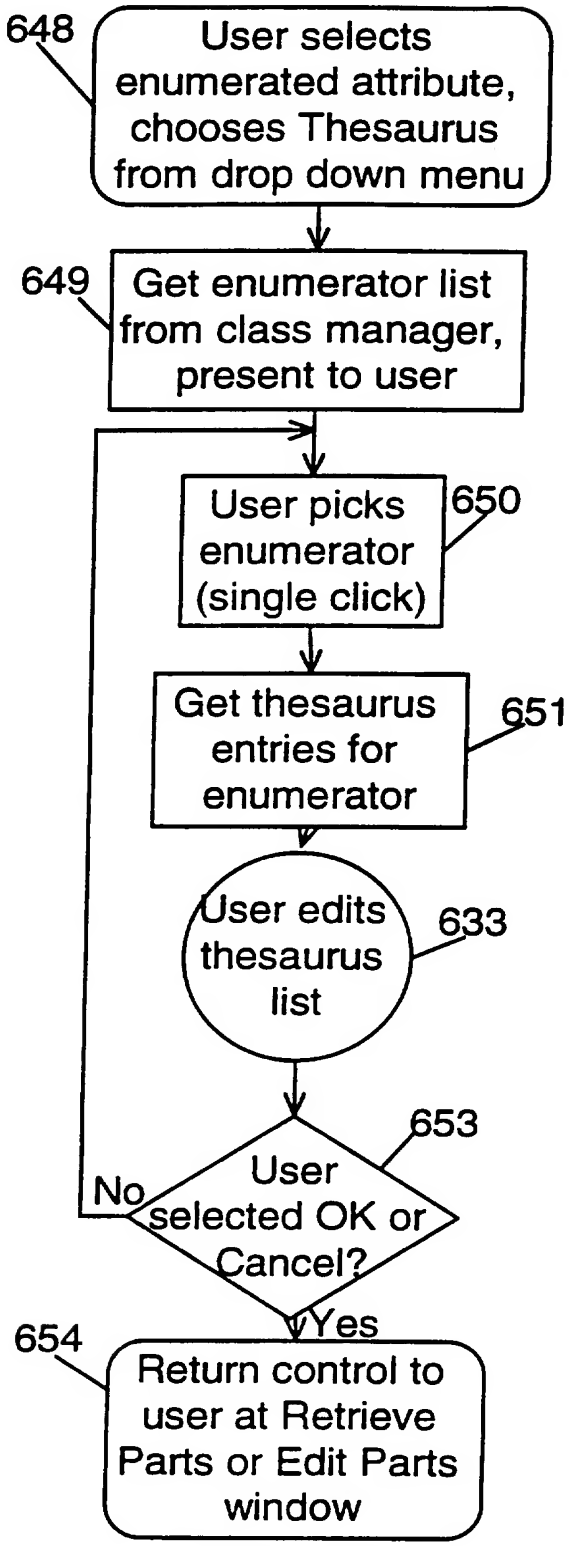

648
User selects enumerated attribute, chooses Thesaurus from drop down menu
649
Get enumerator list from class manager, present to user
User picks enumerator (single click)
650
Get thesaurus entries for enumerator
651
User edits thesaurus list
633
653
User selected OK or Cancel?
No
Yes
654
Return control to user at Retrieve Parts or Edit Parts window

FIG. 184

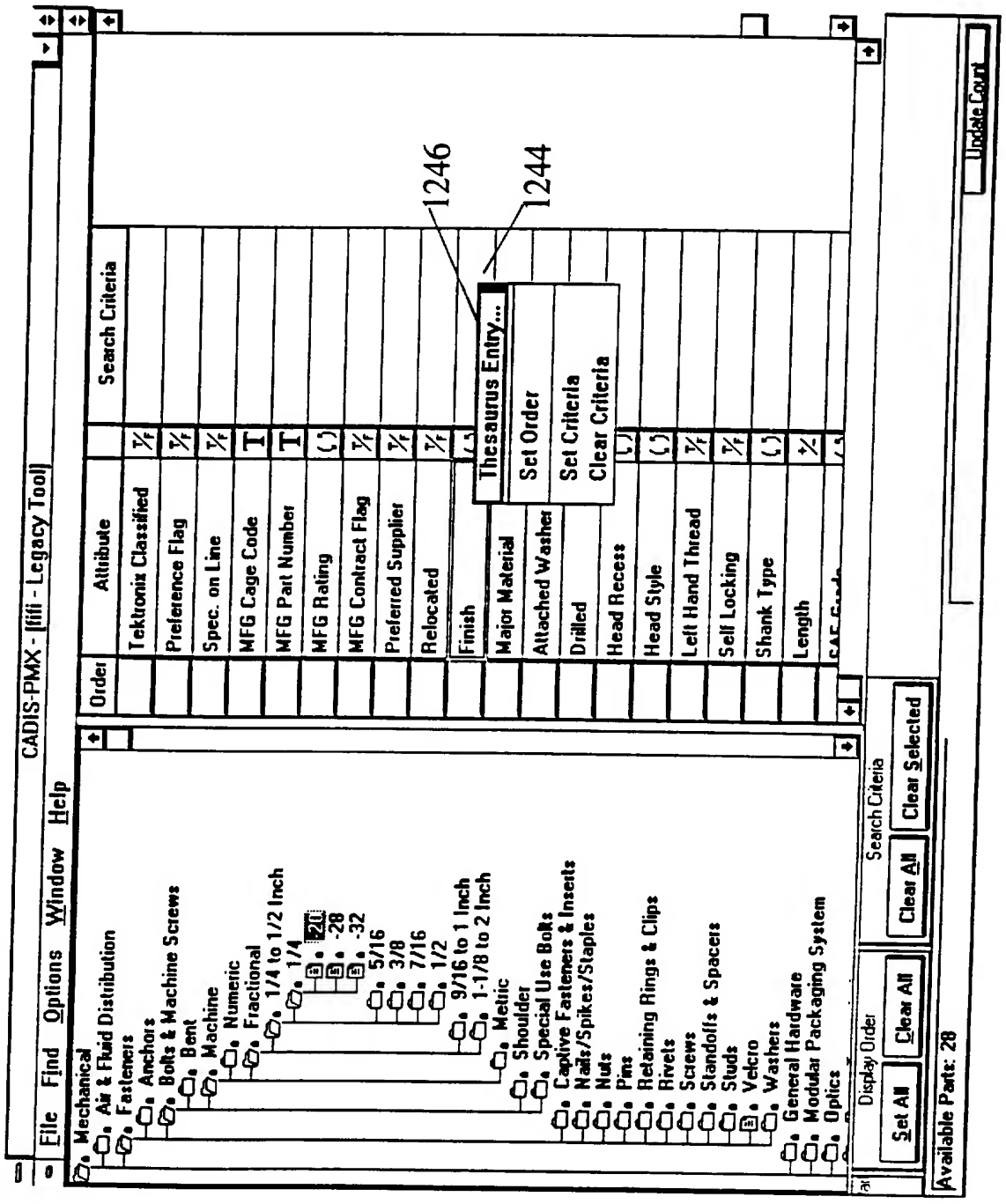

CADIS-PMX - [fifi - Legacy Tool]
File Find Options Window Help
Mechanical
Air & Fluid Distribution
Fasteners
Anchors
Bolts & Machine Screws
Bent
Machine
Numeric
Fractional
1/4 to 1/2 Inch
1/4
#20
-28
-32
5/16
3/8
7/16
1/2
9/16 to 1 Inch
1-1/8 to 2 Inch
Metric
Shoulder
Special Use Bolts
Captive Fasteners & Inserts
Nails/Spikes/Staples
Nuts
Pins
Retaining Rings & Clips
Rivets
Screws
Standoffs & Spacers
Studs
Velcro
Washers
General Hardware
Modular Packaging System
Optics
Order
Attribute
Search Criteria
Tektronix Classified
Preference Flag
Spec. on Line
MFG Cage Code
MFG Part Number
MFG Rating
MFG Contract Flag
Preferred Supplier
Relocated
Finish
Major Material
Attached Washer
Drilled
Head Recess
Head Style
Left Hand Thread
Self Locking
Shank Type
Length
Thesaurus Entry...
Set Order
Set Criteria
Clear Criteria
1246
1244
Update Count
Search Criteria
Clear All
Clear Selected
Display Order
Set All
Clear All
Available Parts: 28

FIG. 185

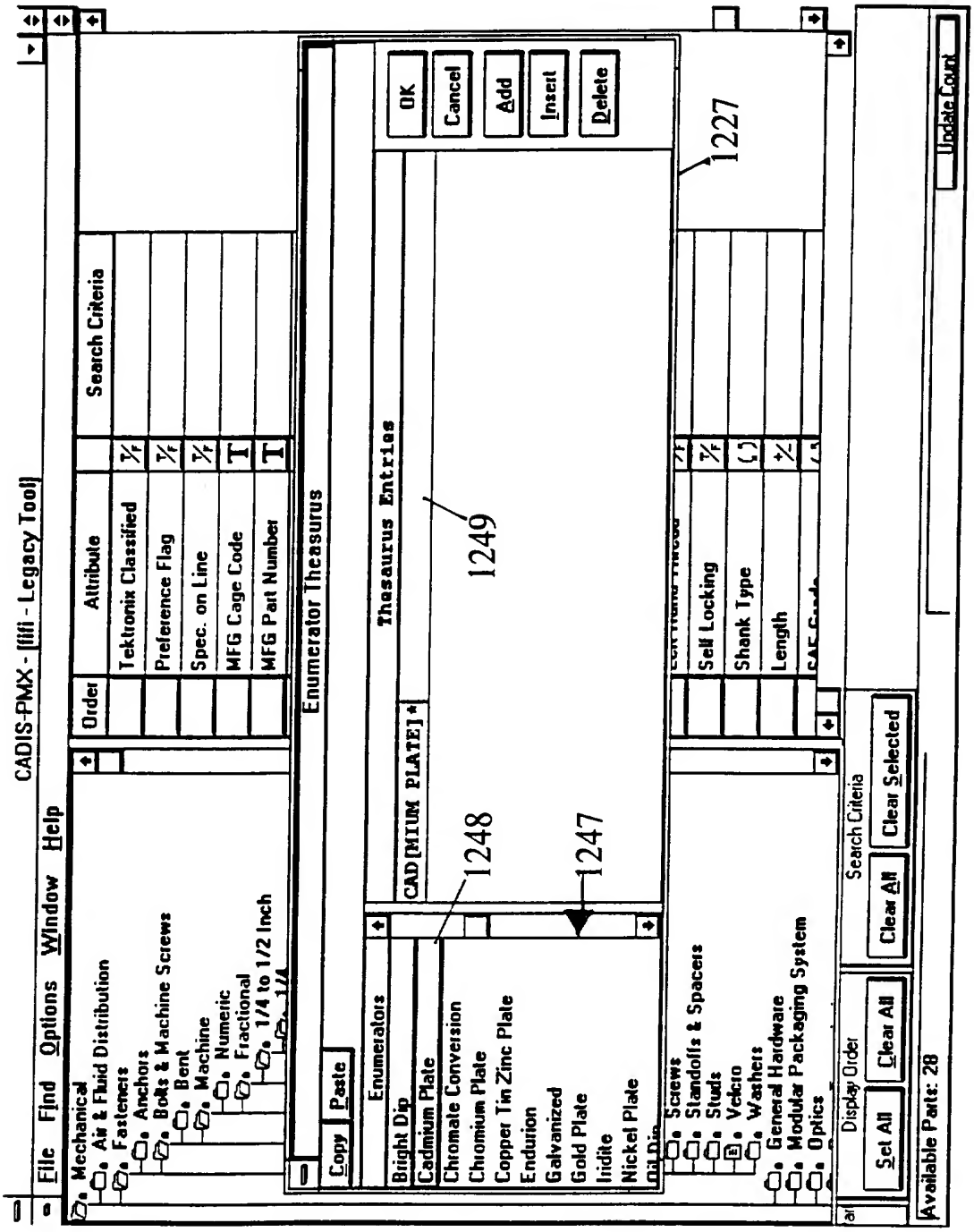

CADIS-PMX - [Ifli - Legacy Tool]
File Find Options Window Help
Mechanical
Air & Fluid Distribution
Fastenecs
Anchors
Bolts & Machine Screws
Bent
Machine
Numeric
Fractional
1/4 to 1/2 Inch
Enumerator Thesaurus
Copy Paste
Enumerators
Bright Dip
Cadmium Plate
Chromate Conversion
Chromium Plate
Copper Tin Zinc Plate
Endurion
Galvanized
Gold Plate
Iridite
Nickel Plate
Order
Attribute
Search Criteria
Tektronix Classified
Preference Flag
Spec. on Line
MFG Cage Code
MFG Part Number
Thesaurus Entries
CAD[MIUM PLATE]
OK
Cancel
Add
Insert
Delete
1249
1248
1247
1227
Self Locking
Shank Type
Length
Screws
Standoffs & Spacers
Studs
Velcro
Washers
General Hardware
Modular Packaging System
Optics
Display Order
Set All
Clear All
Search Criteria
Clear All
Clear Selected
Update Count
Available Parts: 28

FIG. 186

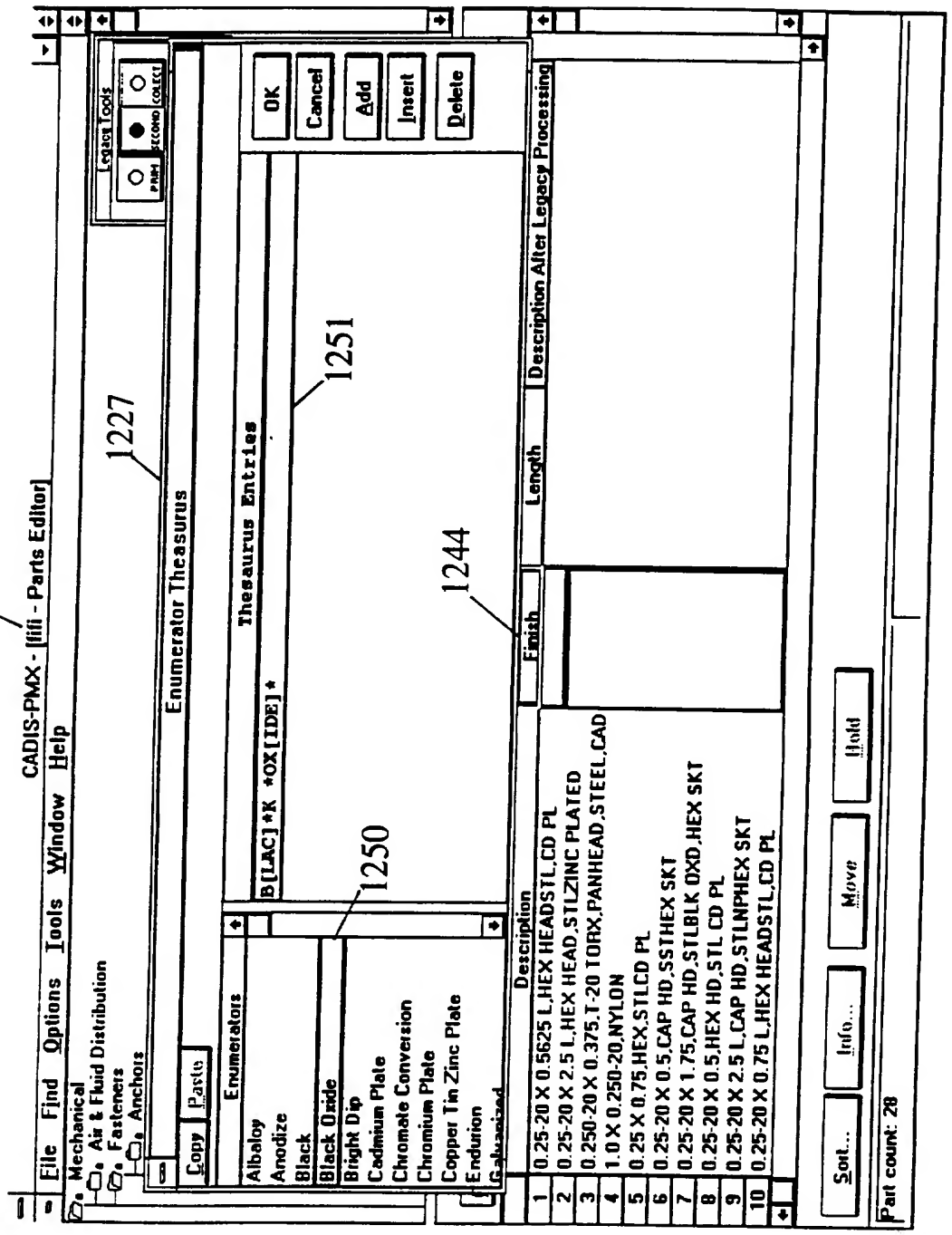

1243
CADIS-PMX - [fifi - Parts Editor]
File Find Options Tools Window Help
Mechanical
Air & Fluid Distribution
Fasteners
Anchors
Legacy Tools
Enumerator Thesaurus
Copy Parts
Enumerators
Albaloy
Anodize
Black
Black Oxide
Bright Dip
Cadmium Plate
Chromate Conversion
Chromium Plate
Copper Tin Zinc Plate
Endurion
Galvanized
1227
Thesaurus Entries
B[LAC]*K *OX[IDE]*
1250
1251
OK
Cancel
Add
Insert
Delete
1244
Finish
Length
Description After Legacy Processing
Description
1 0.25-20 X 0.5625 L,HEX HEADSTL,CD PL
2 0.25-20 X 2.5 L,HEX HEAD,STLZINC PLATED
3 0.250-20 X 0.375,T-20 TORX,PANHEAD,STEEL,CAD
4 1.0 X 0.250-20,NYLON
5 0.25 X 0.75,HEX,STLCD PL
6 0.25-20 X 0.5,CAP HD,SSTHEX SKT
7 0.25-20 X 1.75,CAP HD,STLBLK OXD,HEX SKT
8 0.25-20 X 0.5,HEX HD,STL CD PL
9 0.25-20 X 2.5 L,CAP HD,STLNPHEX SKT
10 0.25-20 X 0.75 L,HEX HEADSTL,CD PL
Sort...
Info...
Move
Hold
Part count: 28

FIG. 187

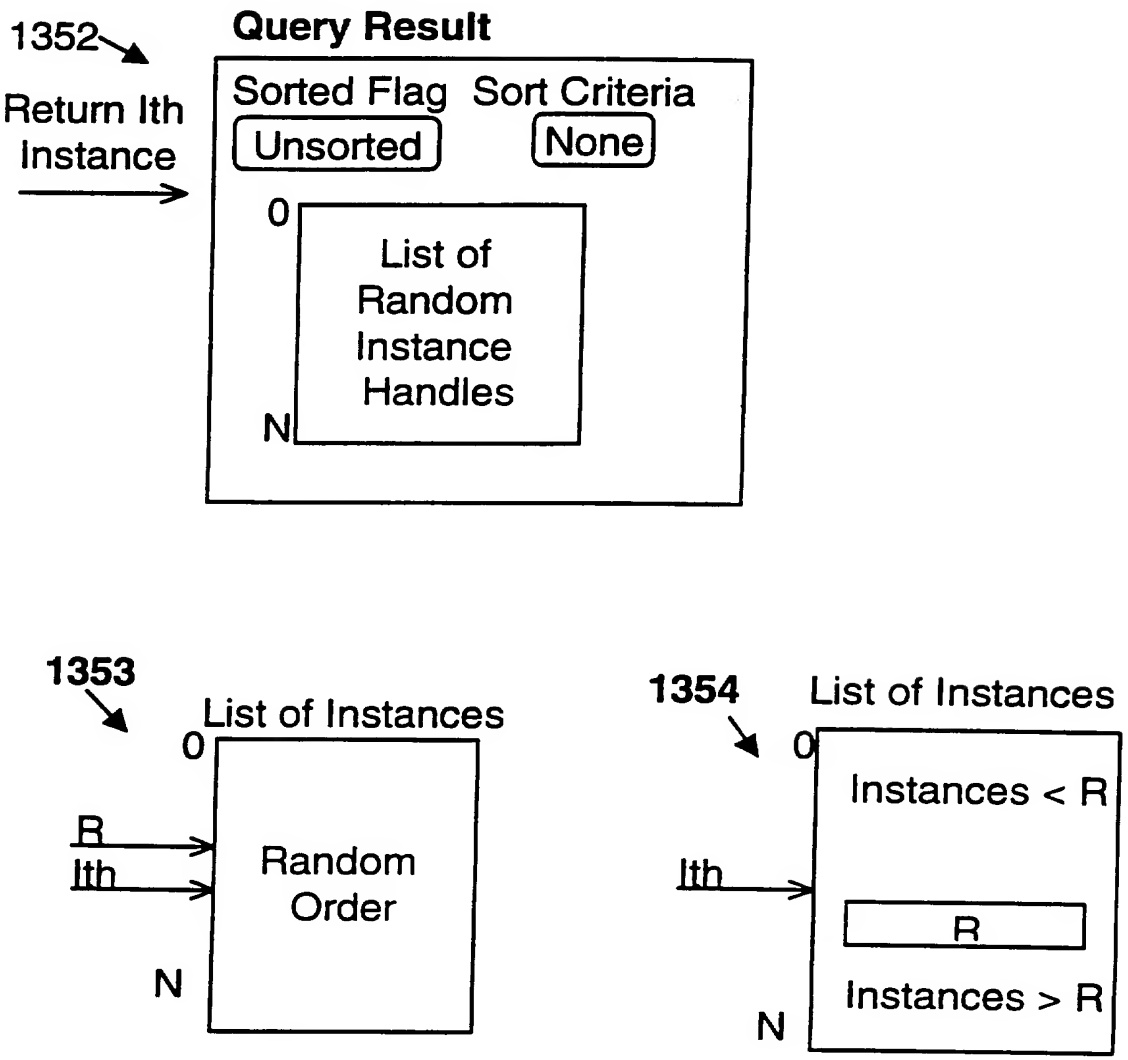

1352
Query Result
Return Ith Instance
Sorted Flag
Sort Criteria
Unsorted
None
0
List of Random Instance Handles
N
1353
List of Instances
0
R
Ith
Random Order
N
1354
List of Instances
0
Instances < R
Ith
R
Instances > R
N

FIG. 188

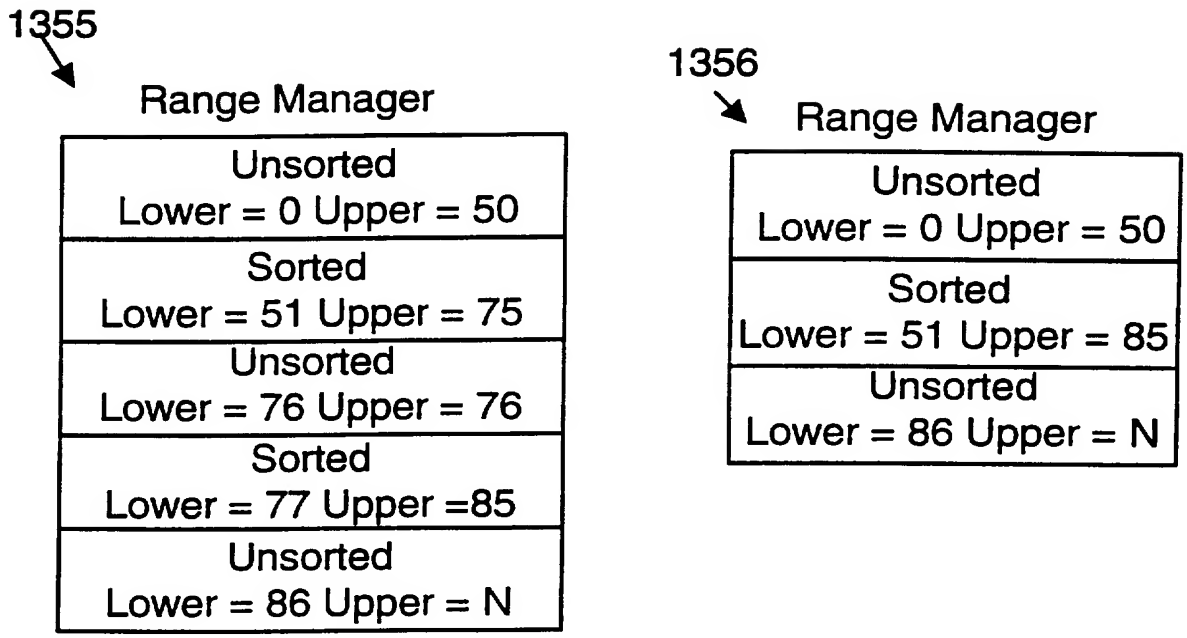

1355
Range Manager
Unsorted
Lower = 0 Upper = 50
Sorted
Lower = 51 Upper = 75
Unsorted
Lower = 76 Upper = 76
Sorted
Lower = 77 Upper =85
Unsorted
Lower = 86 Upper = N
1356
Range Manager
Unsorted
Lower = 0 Upper = 50
Sorted
Lower = 51 Upper = 85
Unsorted
Lower = 86 Upper = N

FIG. 189

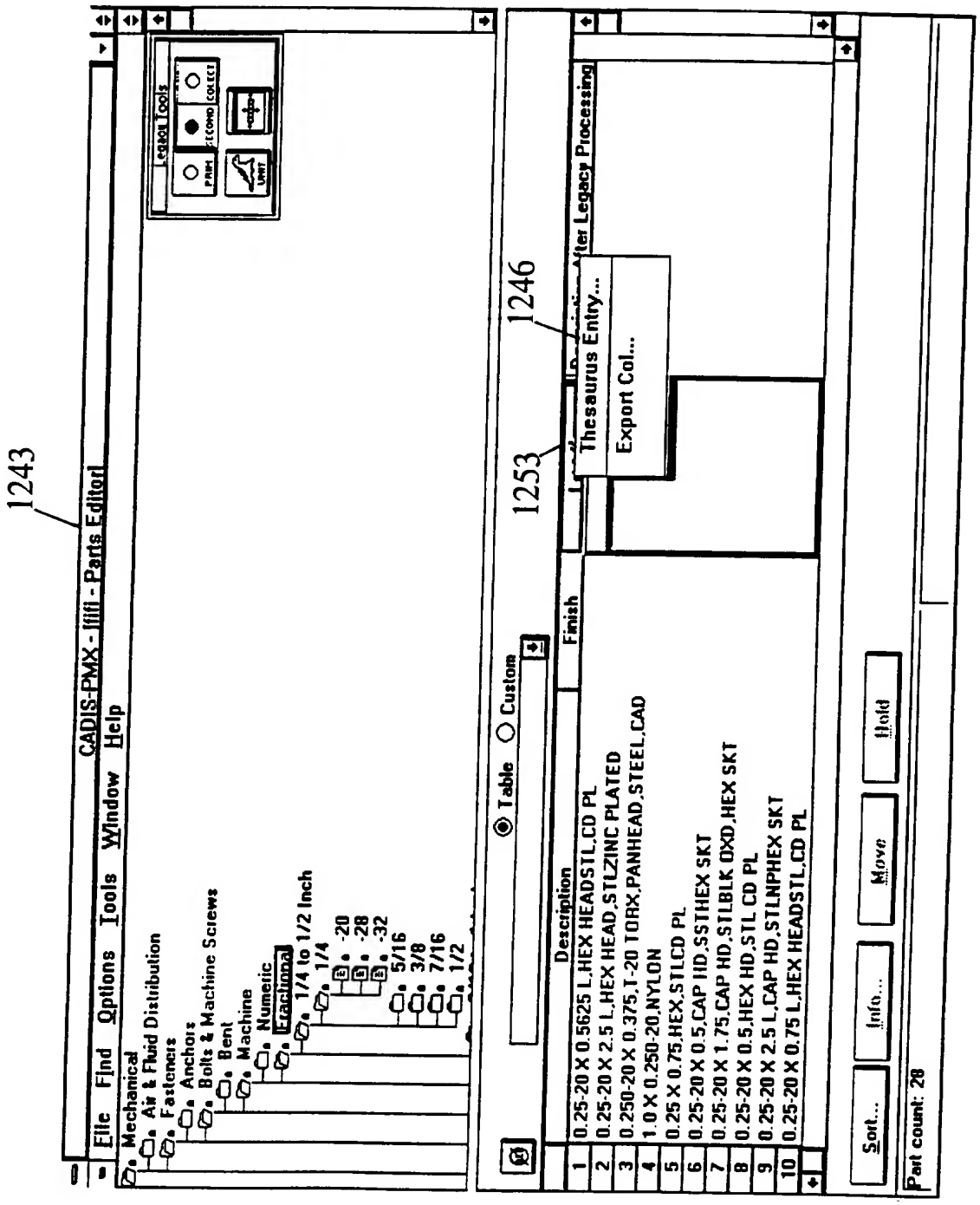

1243
1246
1253
CADIS-PMX - Jiiii - Parts Editor
File Find Options Tools Window Help
Mechanical
Air & Fluid Distribution
Fasteners
Anchors
Bolts & Machine Screws
Bent
Machine
Numeric
Fractional
1/4 to 1/2 Inch
1/4
-20
-28
-32
5/16
3/8
7/16
1/2
Legacy Tools
Table
Custom
Finish
Description
0.25-20 X 0.5625 L,HEX HEADSTL,CD PL
0.25-20 X 2.5 L,HEX HEAD,STLZINC PLATED
0.250-20 X 0.375,T-20 TORX,PANHEAD,STEEL,CAD
1.0 X 0.250-20,NYLON
0.25 X 0.75,HEX,STLCD PL
0.25-20 X 0.5,CAP HD,SSTHEX SKT
0.25-20 X 1.75,CAP HD,STLBLK OXD,HEX SKT
0.25-20 X 0.5,HEX HD,STL CD PL
0.25-20 X 2.5 L,CAP HD,STLNPHEX SKT
0.25-20 X 0.75 L,HEX HEADSTL,CD PL
Thesaurus Entry...
Export Col...
Sort...
Info...
Move
Hold
Part count: 28

FIG. 190

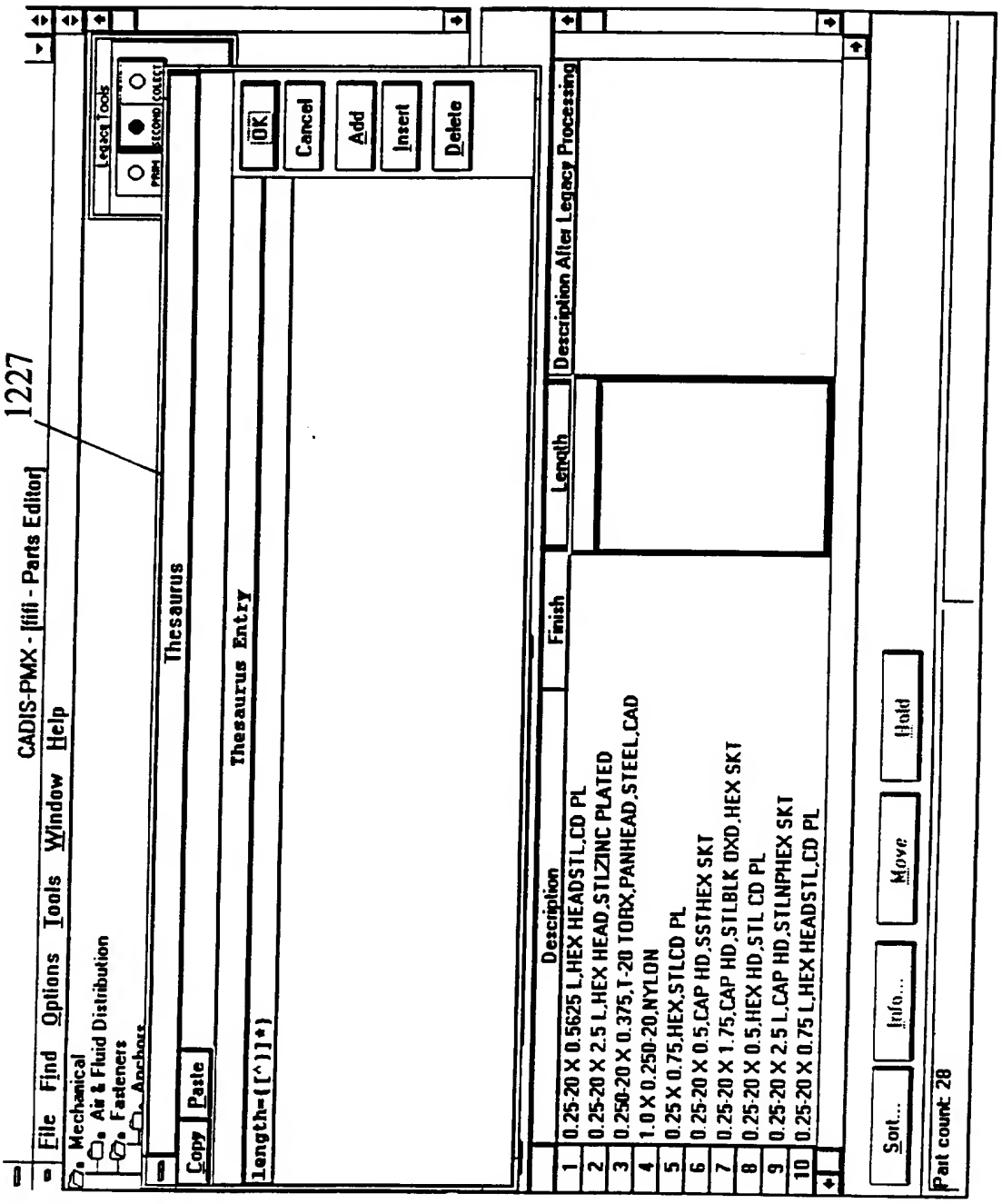

1227
CADIS-PMX - [fifi - Parts Editor]
File Find Options Tools Window Help
Mechanical
Air & Fluid Distribution
Fasteners
Anchors
Thesaurus
Copy Paste
Thesaurus Entry
Length=([^)]*)
Legacy Tools
Prefix
Second
Context
OK
Cancel
Add
Insert
Delete
Description
Finish
Length
Description After Legacy Processing
1 0.25-20 X 0.5625 L,HEX HEADSTL,CD PL
2 0.25-20 X 2.5 L,HEX HEAD,STLZINC PLATED
3 0.250-20 X 0.375,T-20 TORX,PANHEAD,STEEL,CAD
4 1.0 X 0.250-20,NYLON
5 0.25 X 0.75,HEX,STLCD PL
6 0.25-20 X 0.5,CAP HD,SSTHEX SKT
7 0.25-20 X 1.75,CAP HD,STLBLK OXD,HEX SKT
8 0.25-20 X 0.5,HEX HD,STL CD PL
9 0.25-20 X 2.5 L,CAP HD,STLNPHEX SKT
10 0.25-20 X 0.75 L,HEX HEADSTL,CD PL
Sort...
Info...
Move
Hold
Part count: 28

FIG. 191

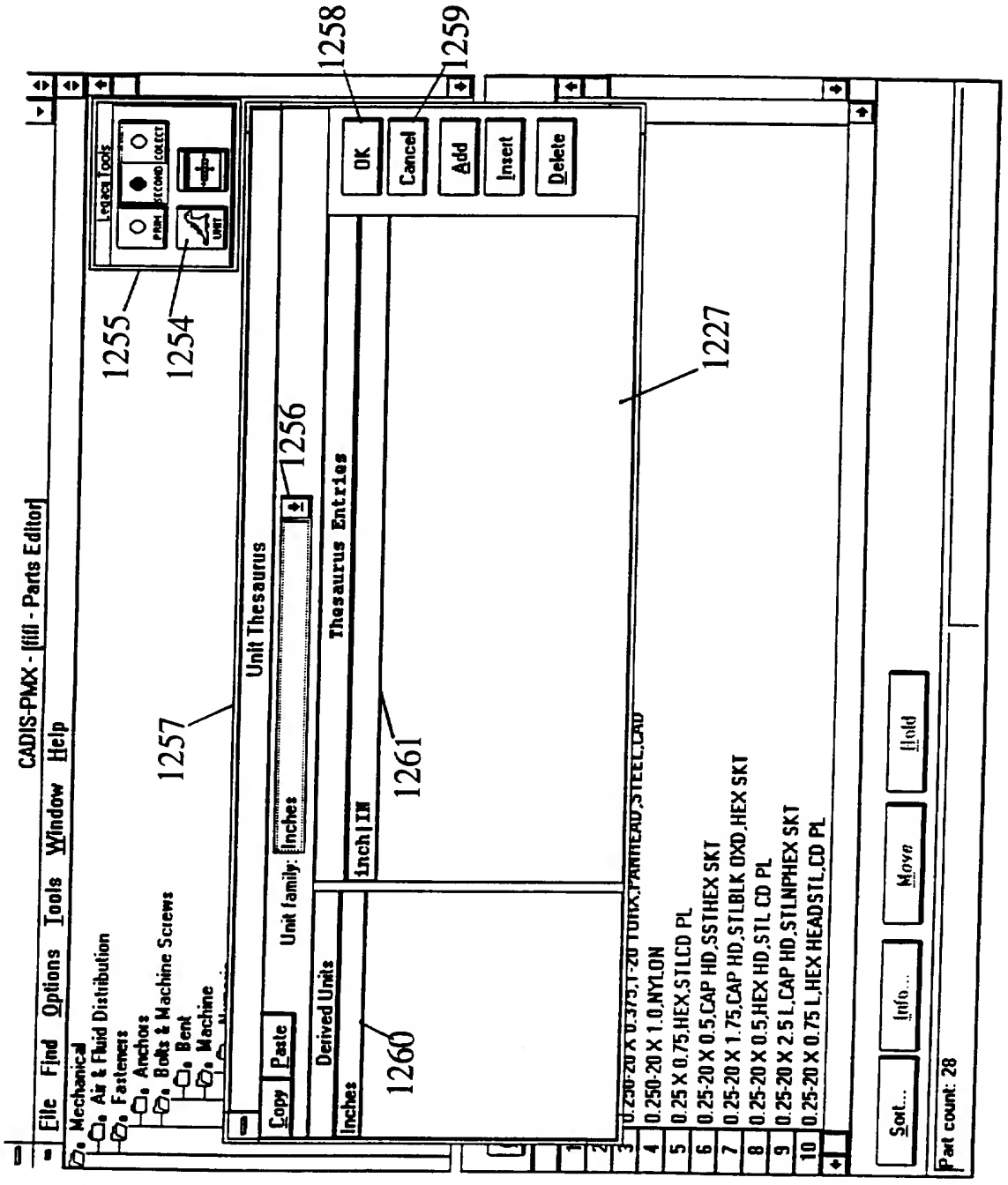

1258
1259
1255
1254
1256
1227
1257
1261
1260
CADIS-PMX - [fifi - Parts Editor]
File Find Options Tools Window Help
Mechanical
Air & Fluid Distribution
Fasteners
Anchors
Bolts & Machine Screws
Bent
Machine
Legacy Tools
PARM
SECOND CORTEC
UNIT
Unit Thesaurus
Unit family: Inches
Copy
Paste
Derived Units
Inches
Thesaurus Entries
Inch | IN
OK
Cancel
Add
Insert
Delete
0.250-20 X 0.375 T-20 TORX,PANHEAD,STEEL,CAD
0.250-20 X 1.0,NYLON
0.25 X 0.75,HEX,STLCD PL
0.25-20 X 0.5,CAP HD,SSTHEX SKT
0.25-20 X 1.75,CAP HD,STLBLK OXD,HEX SKT
0.25-20 X 0.5,HEX HD,STL CD PL
0.25-20 X 2.5 L,CAP HD,STLNPHEX SKT
0.25-20 X 0.75 L,HEX HEADSTL,CD PL
Sort...
Info...
Move
Hold
Part count: 28

FIG. 192

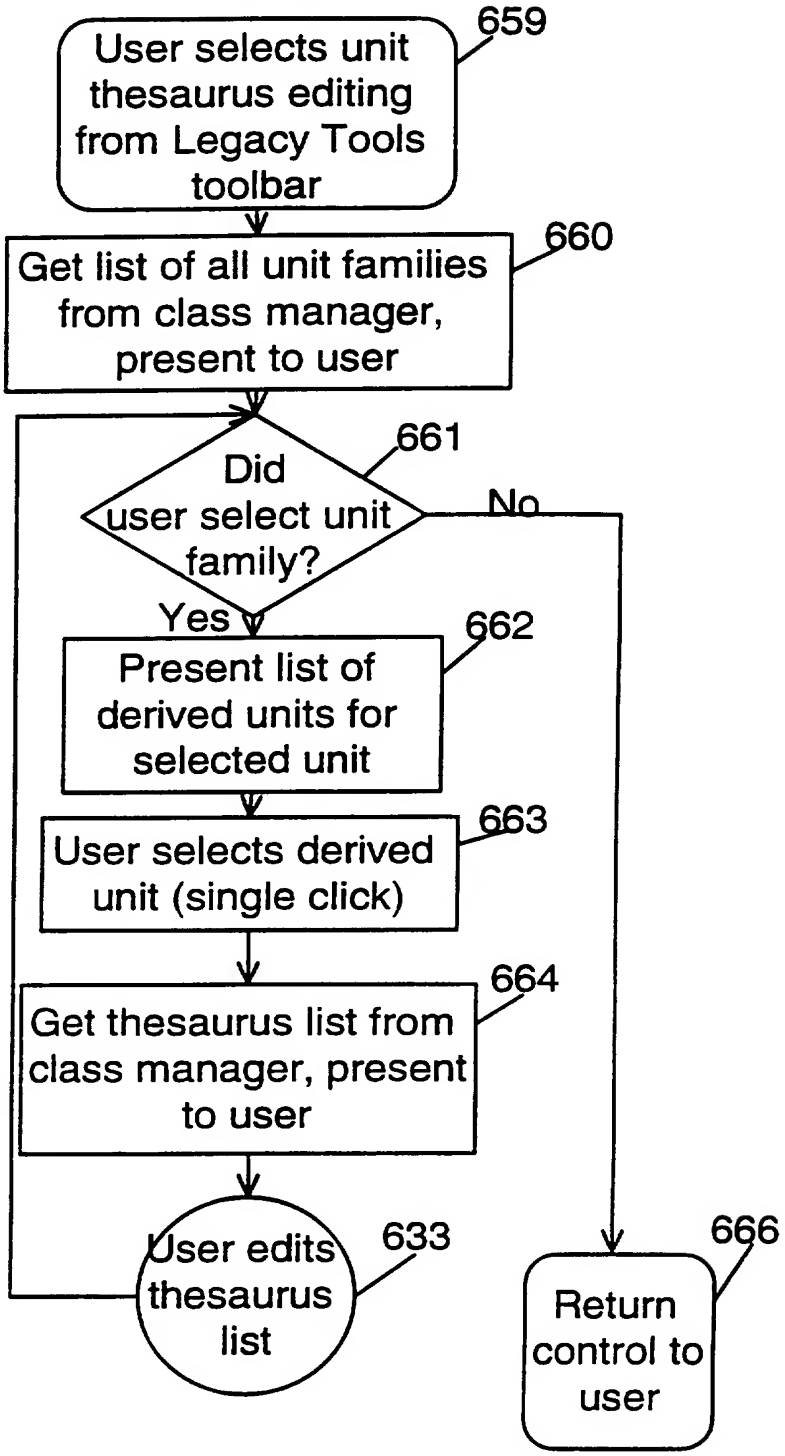

User selects unit thesaurus editing from Legacy Tools toolbar
659
Get list of all unit families from class manager, present to user
660
Did user select unit family?
661
No
Yes
Present list of derived units for selected unit
662
User selects derived unit (single click)
663
Get thesaurus list from class manager, present to user
664
User edits thesaurus list
633
Return control to user
666

FIG. 193

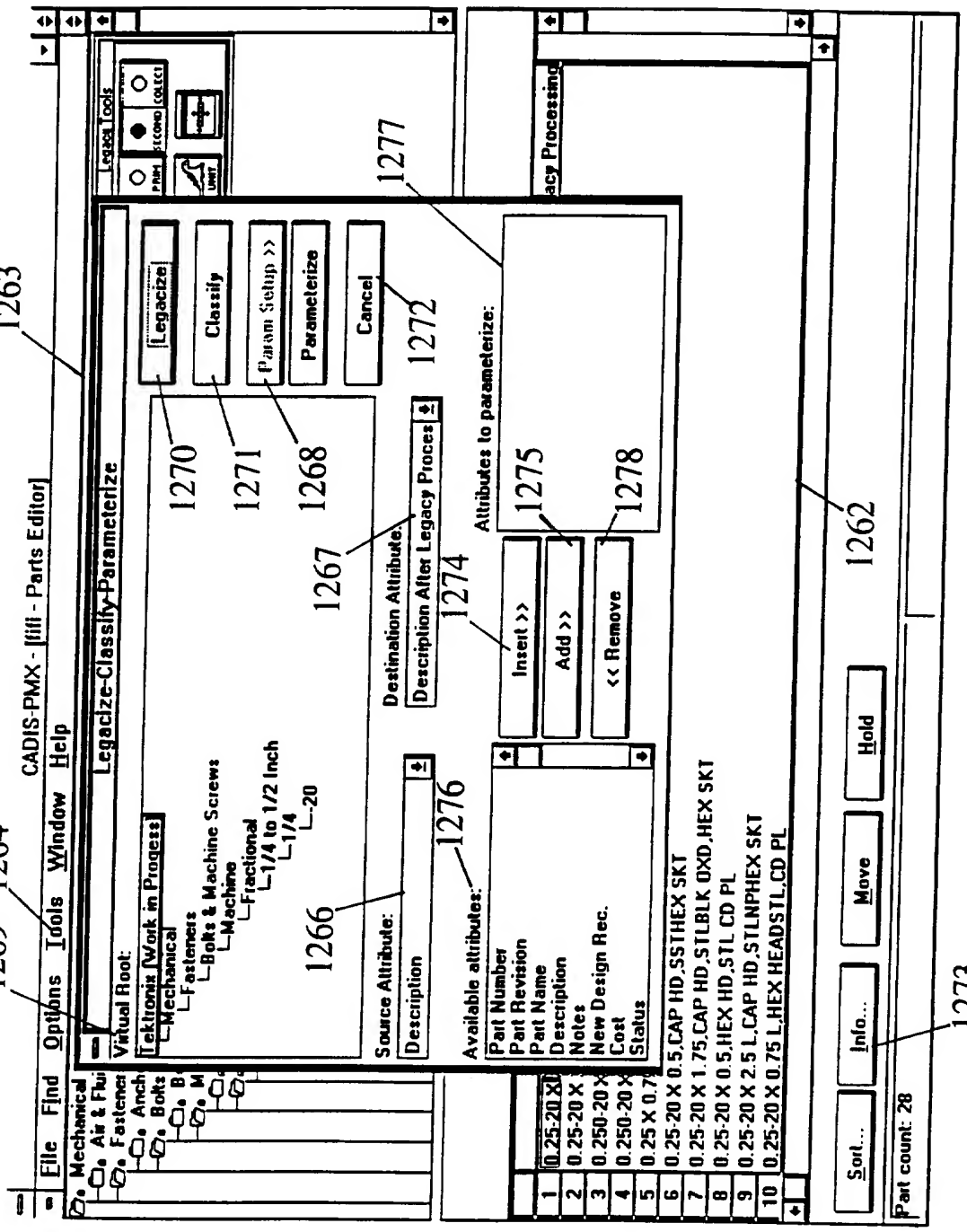

CADIS-PMX - [fifi - Parts Editor]
File Find Options Tools Window Help
Legacize-Classify-Parameterize
Virtual Root:
Tektronix Work in Progress
Mechanical
Fasteners
Bolts & Machine Screws
Machine
Fractional
1/4 to 1/2 Inch
1/4
-20
Legacize
Classify
Param Setup >>
Parameterize
Cancel
Source Attribute:
Description
Destination Attribute:
Description After Legacy Proces
Available attributes:
Part Number
Part Revision
Part Name
Description
Notes
New Design Rec.
Cost
Status
Insert >>
Add >>
<< Remove
Attributes to parameterize:
Sort...
Info...
Move
Hold
Part count: 28
1263
1264
1269
1270
1271
1268
1267
1266
1276
1274
1275
1278
1277
1272
1262
1273

FIG. 194

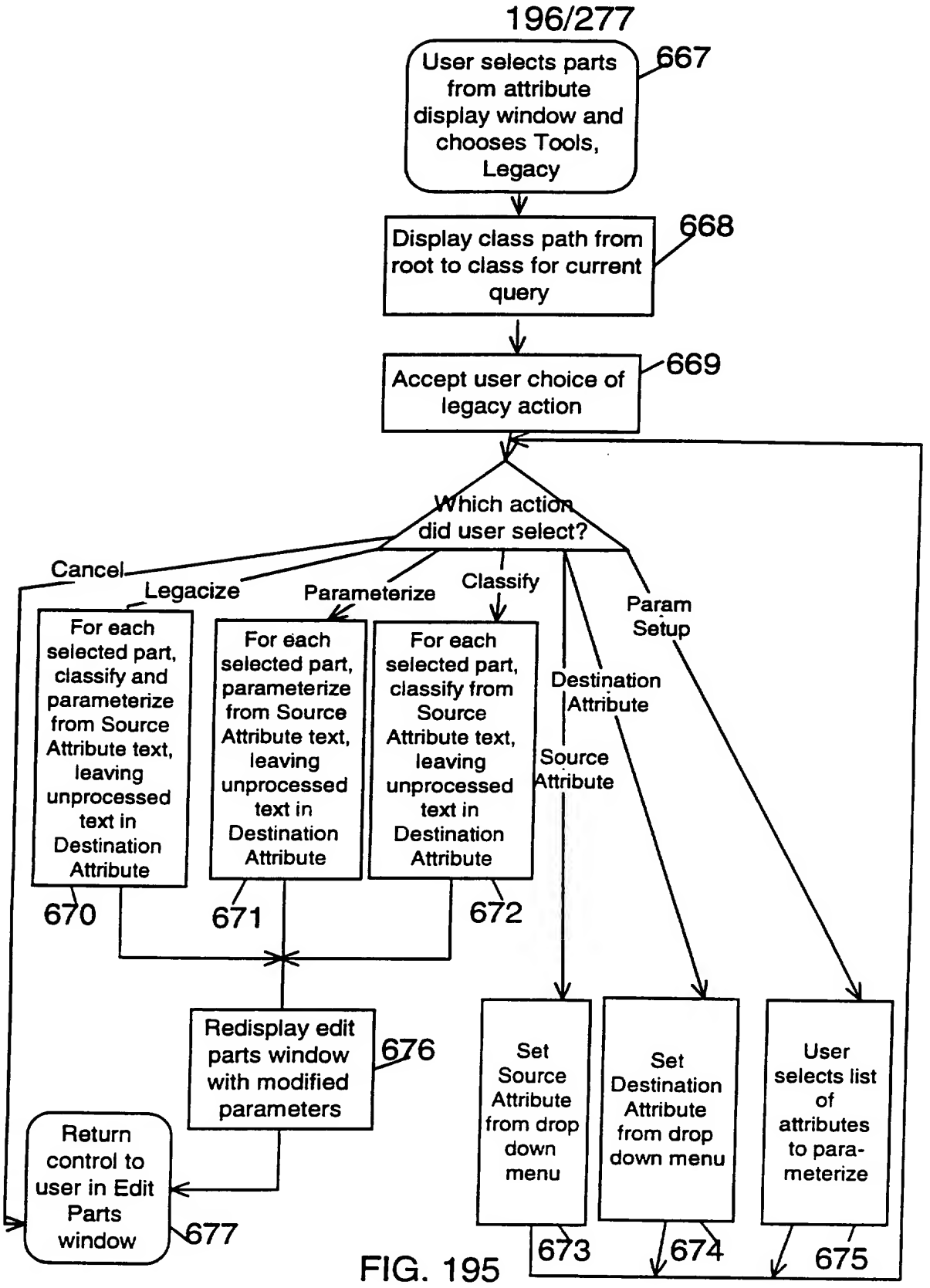

User selects parts from attribute display window and chooses Tools, Legacy
667
Display class path from root to class for current query
668
Accept user choice of legacy action
669
Which action did user select?
Cancel
Legacize
Parameterize
Classify
Param Setup
Destination Attribute
Source Attribute
For each selected part, classify and parameterize from Source Attribute text, leaving unprocessed text in Destination Attribute
For each selected part, parameterize from Source Attribute text, leaving unprocessed text in Destination Attribute
For each selected part, classify from Source Attribute text, leaving unprocessed text in Destination Attribute
670
671
672
Redisplay edit parts window with modified parameters
676
Return control to user in Edit Parts window
677
Set Source Attribute from drop down menu
Set Destination Attribute from drop down menu
User selects list of attributes to para-meterize
673
674
675

FIG. 195

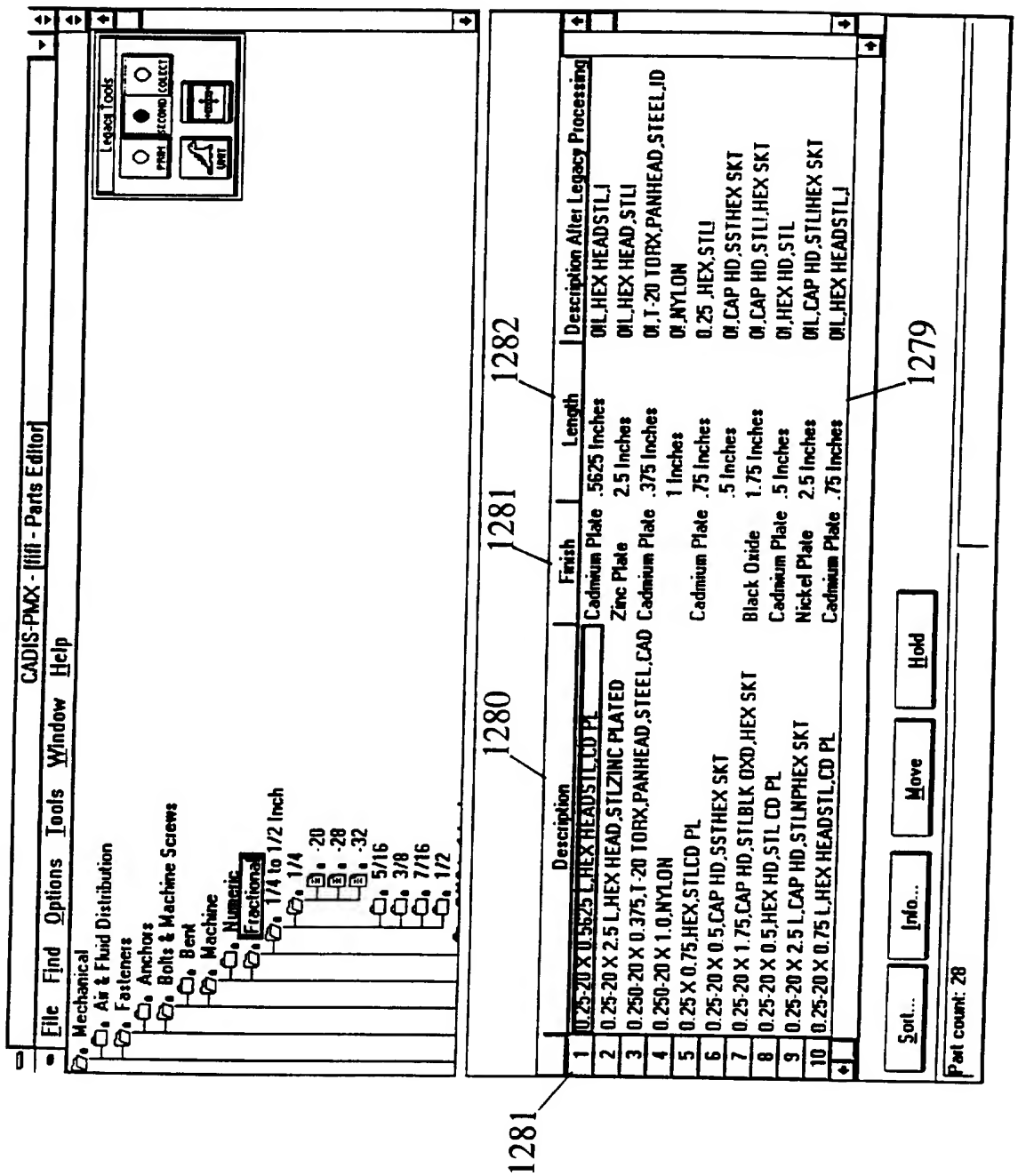

CADIS-PMX - [Iiii - Parts Editor]
File Find Options Tools Window Help
Mechanical
Air & Fluid Distribution
Fasteners
Anchors
Bolts & Machine Screws
Bent
Machine
Numeric
Fractional
1/4 to 1/2 Inch
1/4
-20
-28
-32
5/16
3/8
7/16
1/2
Legacy Tools
PUSH
SECOND
COLLECT
UNIT
Description
Finish
Length
Description After Legacy Processing
1 0.25-20 X 0.5625 L,HEX HEADSTL,CD PL Cadmium Plate .5625 Inches 01L,HEX HEADSTL,I
2 0.25-20 X 2.5 L,HEX HEAD,STLZINC PLATED Zinc Plate 2.5 Inches 01L,HEX HEAD,STLI
3 0.250-20 X 0.375,T-20 TORX,PANHEAD,STEEL,CAD Cadmium Plate .375 Inches 01,T-20 TORX,PANHEAD,STEEL,ID
4 0.250-20 X 1.0,NYLON 1 Inches 01,NYLON
5 0.25 X 0.75,HEX,STLCD PL Cadmium Plate .75 Inches 0.25 ,HEX,STLI
6 0.25-20 X 0.5,CAP HD,SSTHEX SKT .5 Inches 01,CAP HD,SSTHEX SKT
7 0.25-20 X 1.75,CAP HD,STLBLK OXD,HEX SKT Black Oxide 1.75 Inches 01,CAP HD,STLI,HEX SKT
8 0.25-20 X 0.5,HEX HD,STL CD PL Cadmium Plate .5 Inches 01,HEX HD,STL
9 0.25-20 X 2.5 L,CAP HD,STLNPHEX SKT Nickel Plate 2.5 Inches 01L,CAP HD,STLHEX SKT
10 0.25-20 X 0.75 L,HEX HEADSTL,CD PL Cadmium Plate .75 Inches 01L,HEX HEADSTL,I
Sort...
Info...
Move
Hold
Part count: 28
1280
1281
1282
1279
1281

FIG. 196

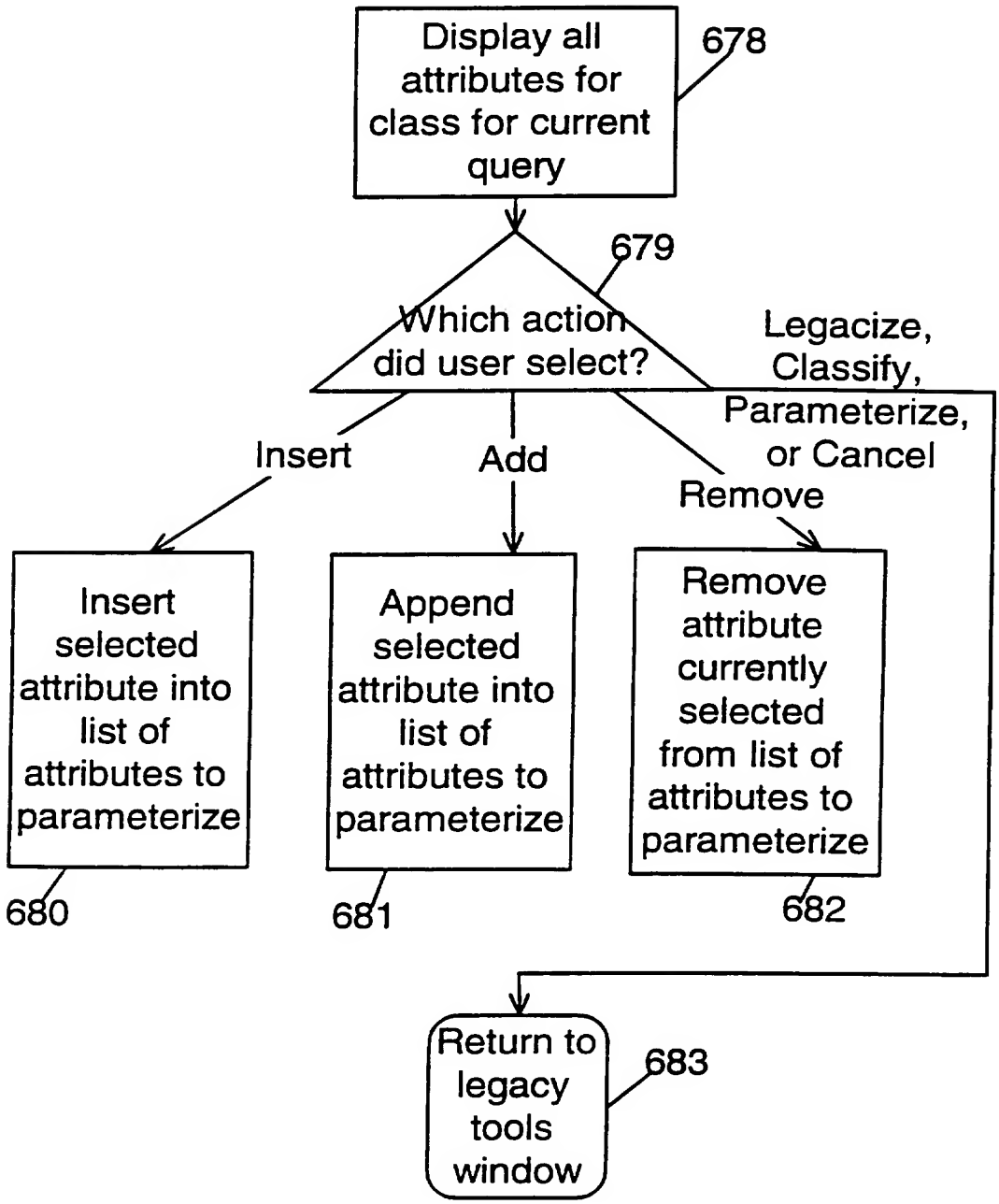

Display all attributes for class for current query
678
679
Which action did user select?
Legacize, Classify, Parameterize, or Cancel
Insert
Add
Remove
Insert selected attribute into list of attributes to parameterize
Append selected attribute into list of attributes to parameterize
Remove attribute currently selected from list of attributes to parameterize
680
681
682
Return to legacy tools window
683

FIG. 197

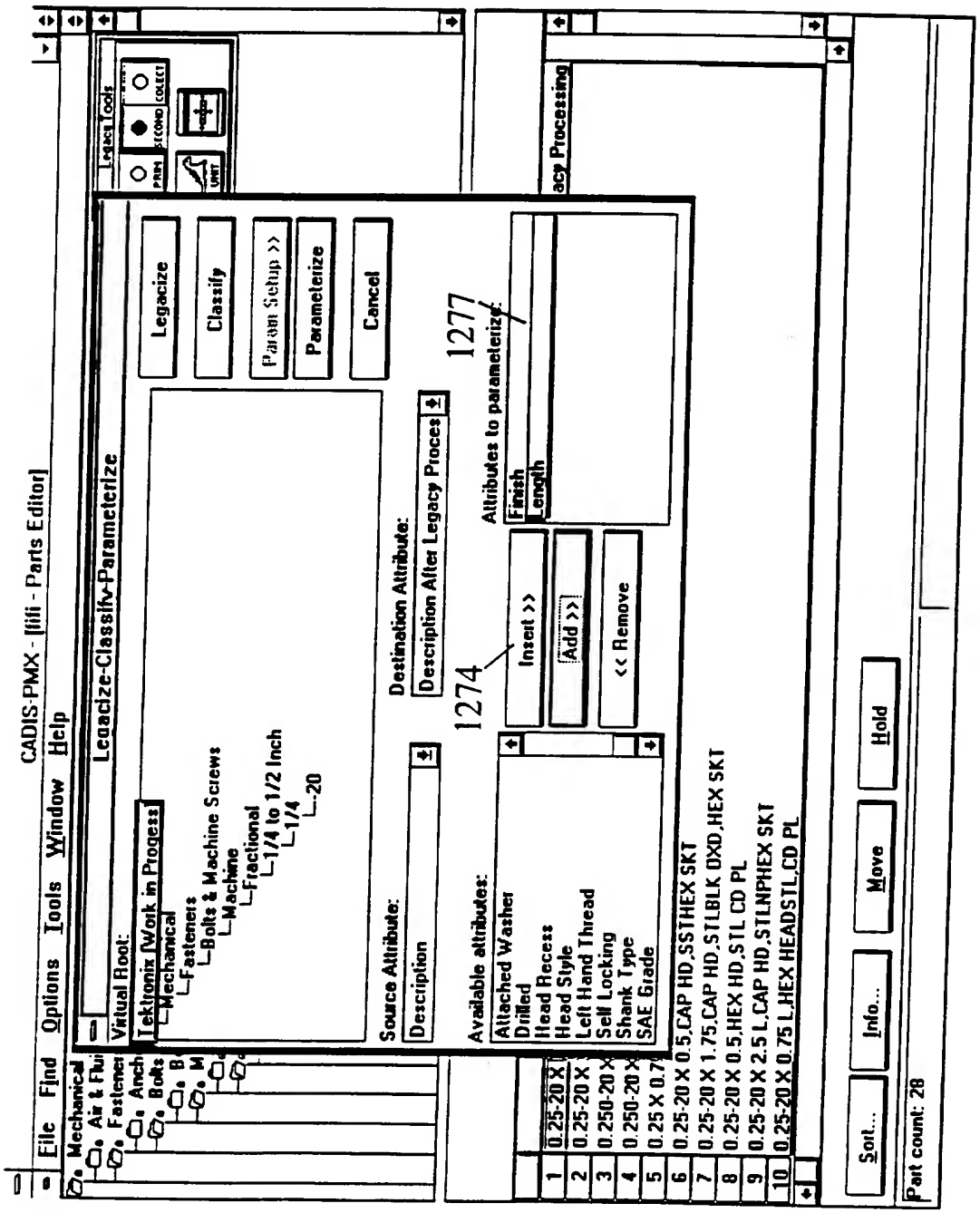

CADIS-PMX - [Iili - Parts Editor]
File Find Options Tools Window Help
Legacize-Classify-Parameterize
Virtual Root:
Tektronix Work in Progess
Mechanical
Fasteners
Bolts & Machine Screws
Machine
Fractional
1/4 to 1/2 Inch
1/4
-20
Legacize
Classify
Param Setup >>
Parameterize
Cancel
Source Attribute:
Description
Destination Attribute:
Description After Legacy Proces
Available attributes:
Attached Washer
Drilled
Head Recess
Head Style
Left Hand Thread
Self Locking
Shank Type
SAE Grade
Insert >>
Add >>
<< Remove
Attributes to parameterize:
Finish
Length
1274
1277
0.25-20 X 0.5,CAP HD,SSTHEX SKT
0.25-20 X 1.75,CAP HD,STLBLK OXD,HEX SKT
0.25-20 X 0.5,HEX HD,STL CD PL
0.25-20 X 2.5 L,CAP HD,STLNPHEX SKT
0.25-20 X 0.75 L,HEX HEADSTL,CD PL
Sort...
Info...
Move
Hold
Part count: 28

FIG. 198

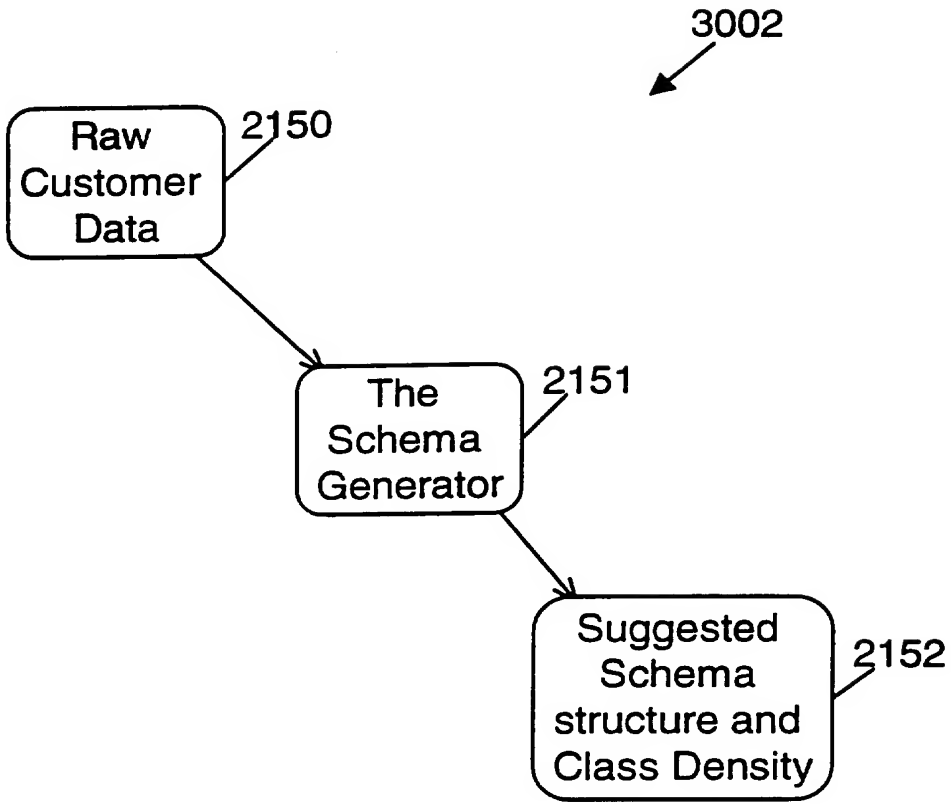

3002
Raw Customer Data
2150
The Schema Generator
2151
Suggested Schema structure and Class Density
2152

FIG. 199

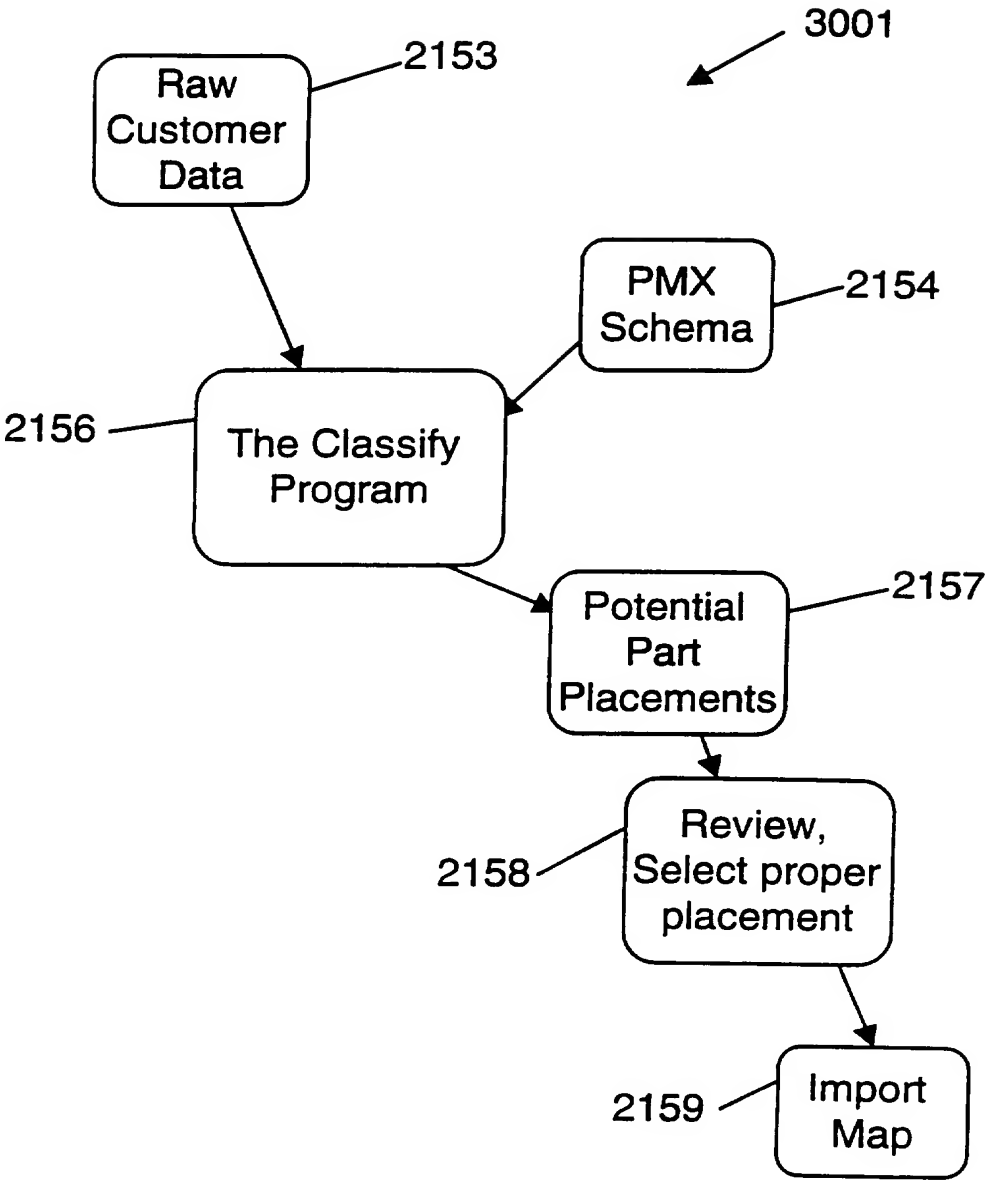

3001
2153
Raw Customer Data
PMX Schema
2154
2156
The Classify Program
Potential Part Placements
2157
Review, Select proper placement
2158
Import Map
2159

FIG. 200

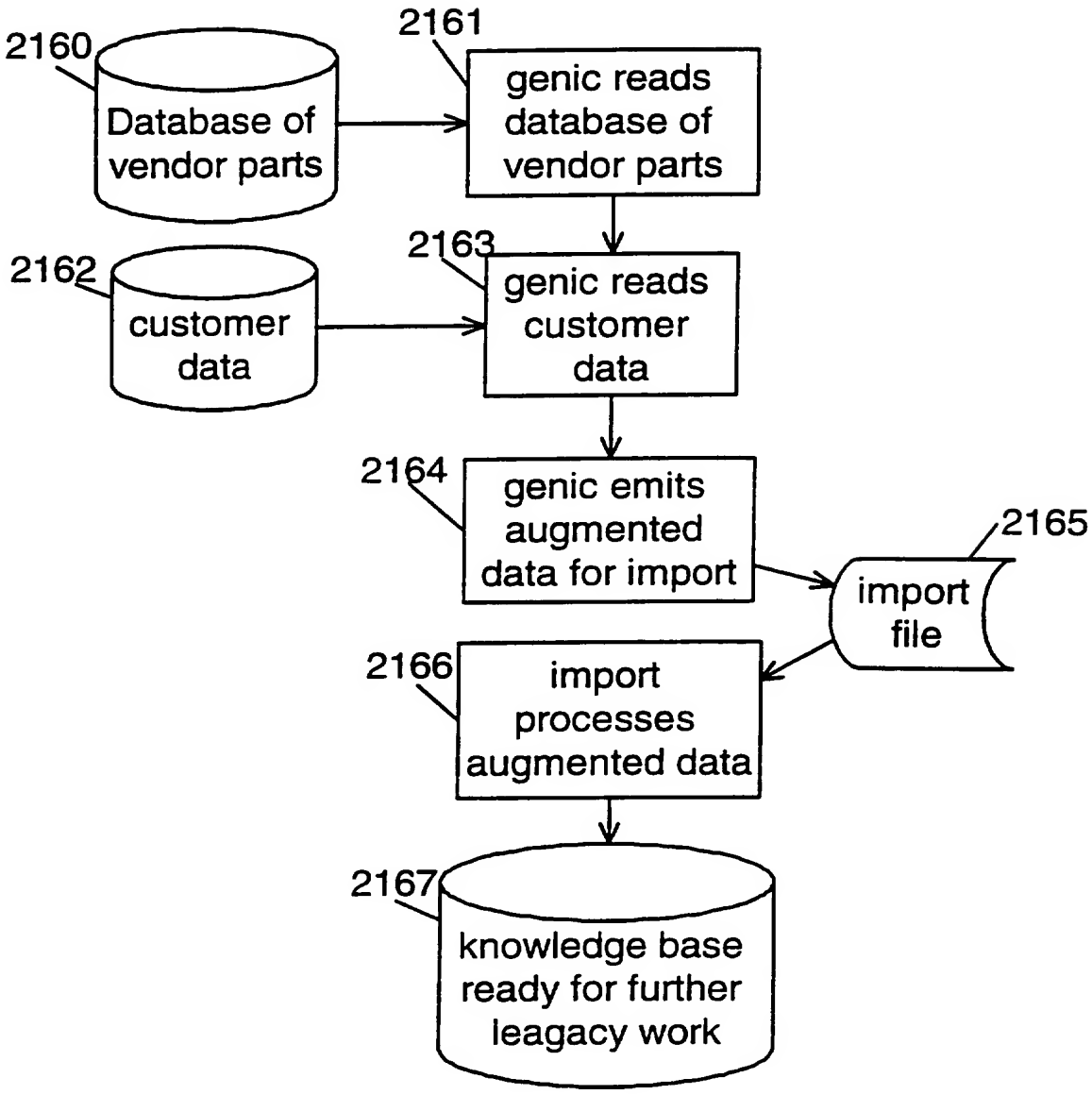

2160
Database of vendor parts
2161
genic reads database of vendor parts
2162
customer data
2163
genic reads customer data
2164
genic emits augmented data for import
2165
import file
2166
import processes augmented data
2167
knowledge base ready for further leagacy work

FIG. 201

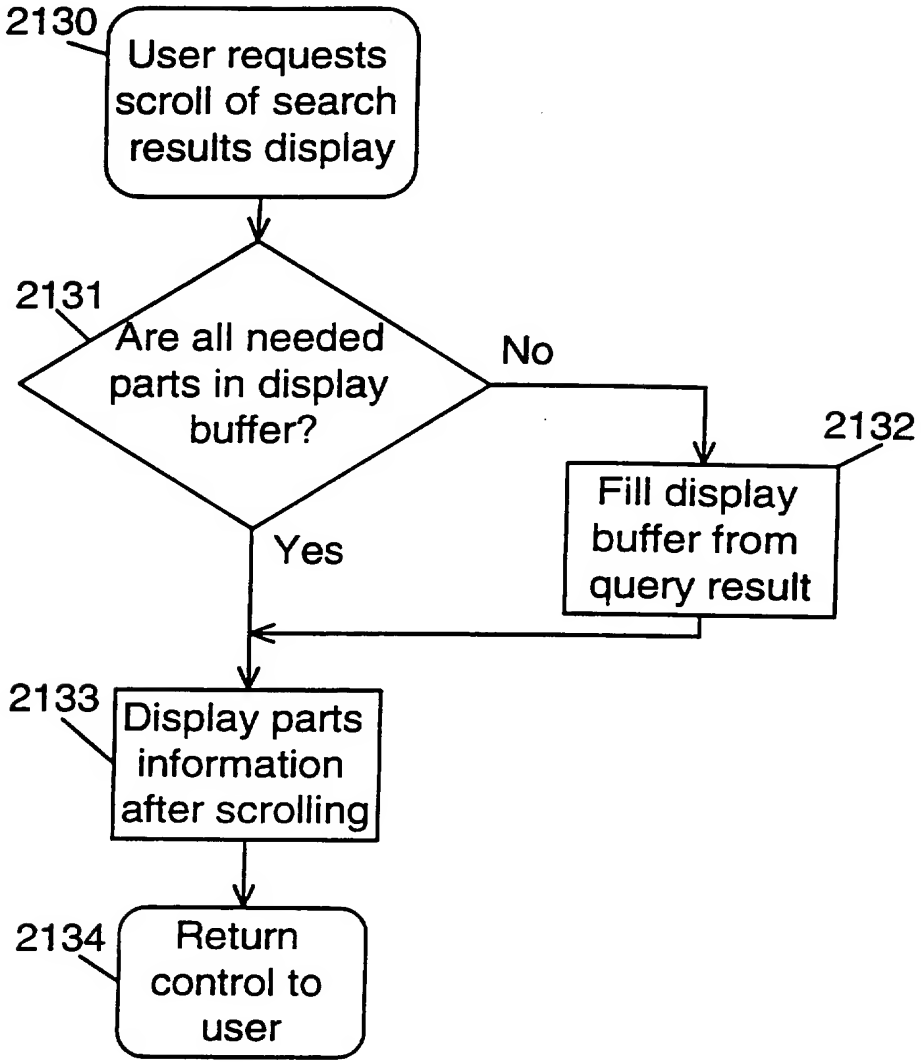

2130
User requests scroll of search results display
2131
Are all needed parts in display buffer?
No
Yes
2132
Fill display buffer from query result
2133
Display parts information after scrolling
2134
Return control to user

FIG. 202

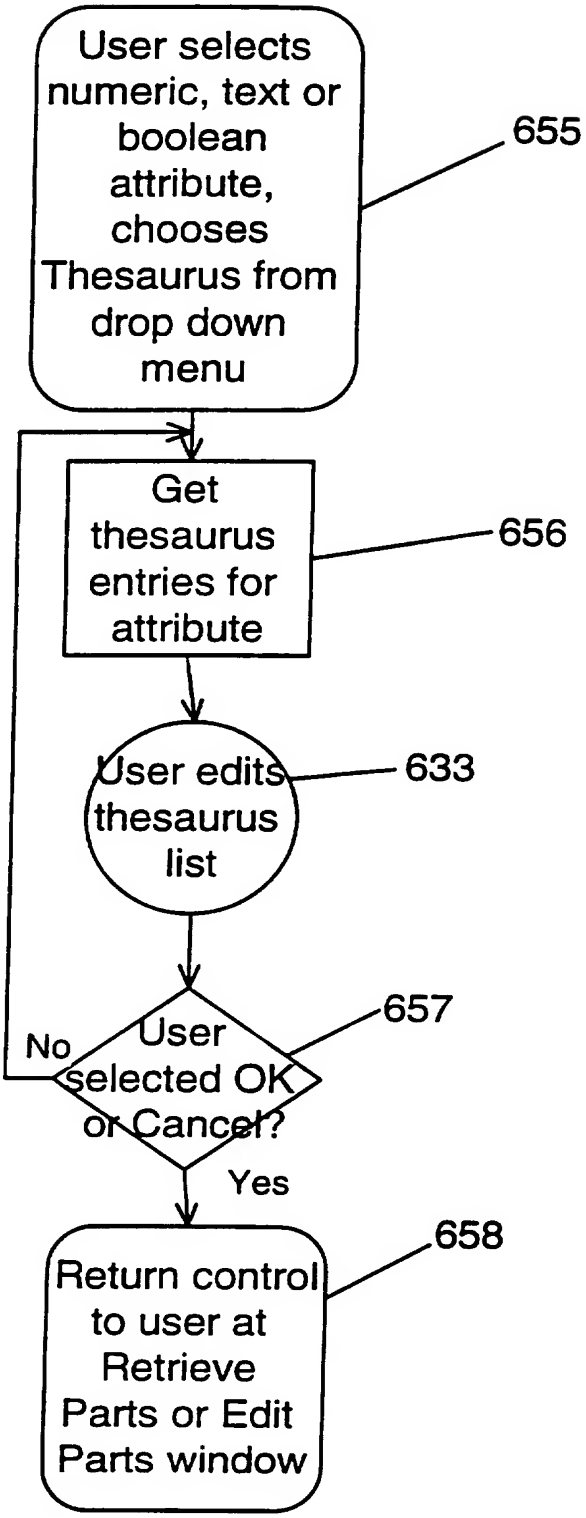

User selects numeric, text or boolean attribute, chooses Thesaurus from drop down menu
655
Get thesaurus entries for attribute
656
User edits thesaurus list
633
User selected OK or Cancel?
657
No
Yes
Return control to user at Retrieve Parts or Edit Parts window
658

FIG. 203

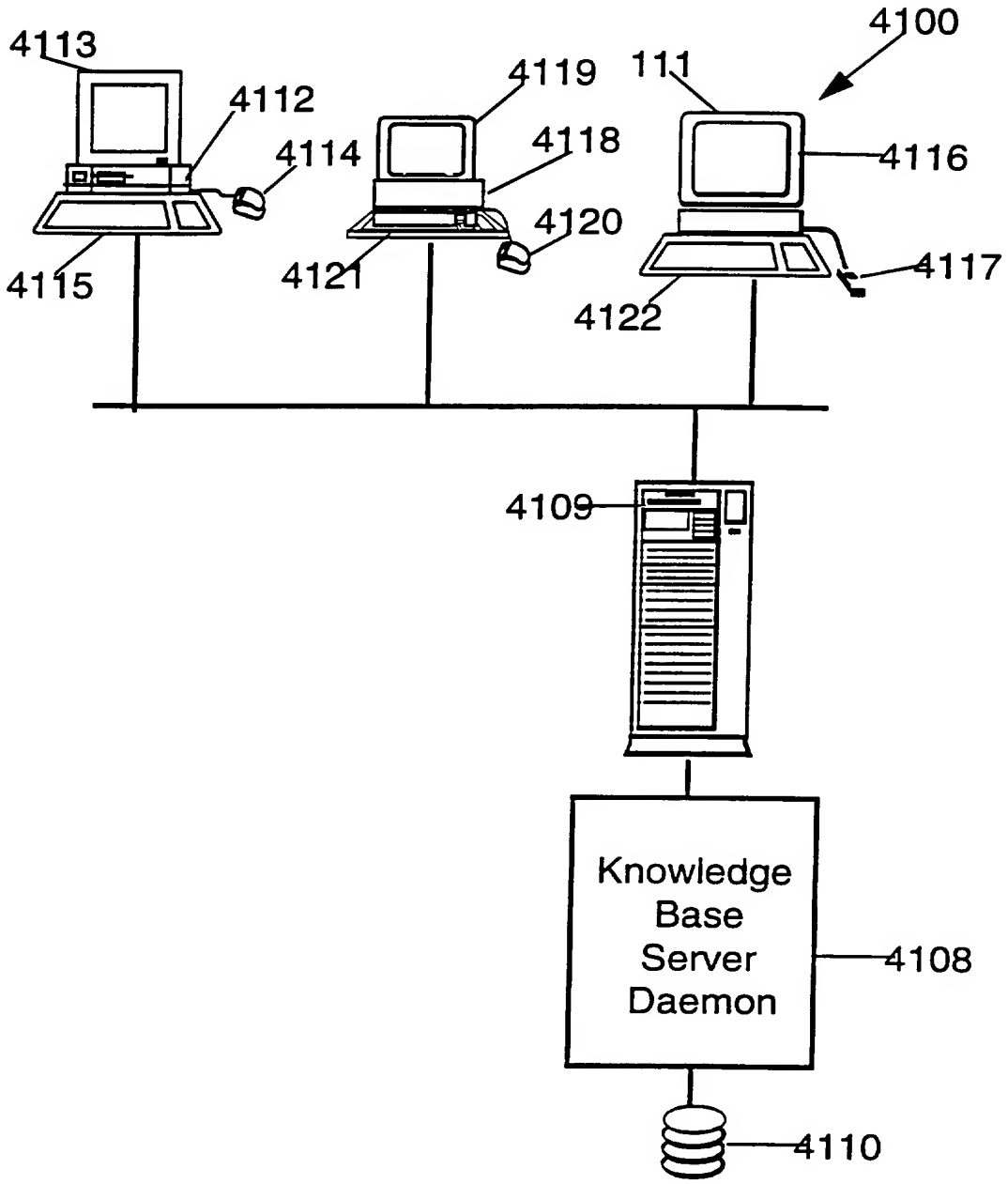

4113
4112
4114
4115
4119
4118
4120
4121
111
4100
4116
4117
4122
4109
Knowledge Base Server Daemon
4108
4110

FIG. 204

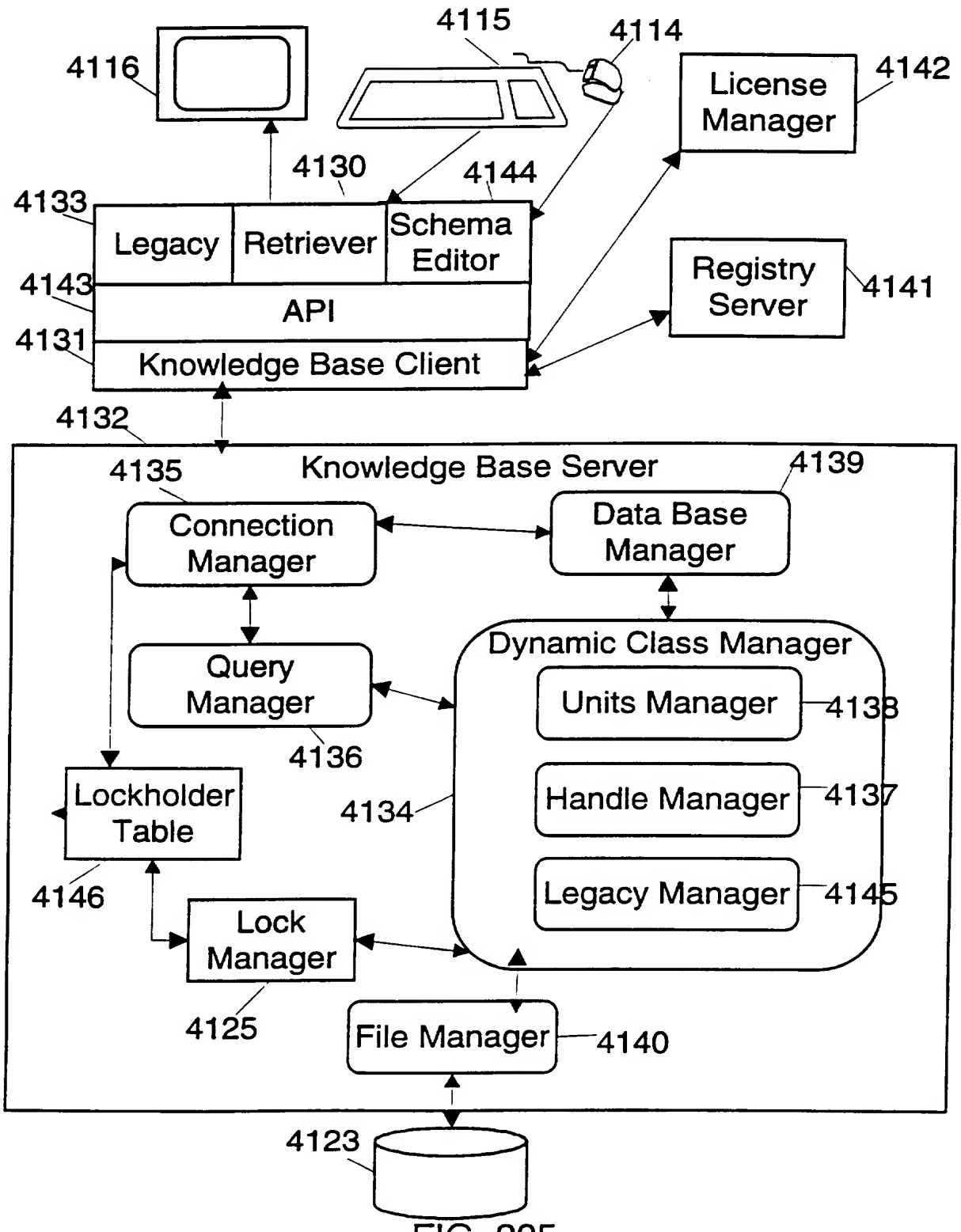

4115
4114
4116
License Manager
4142
4130
4144
4133
Legacy
Retriever
Schema Editor
4143
API
Registry Server
4141
4131
Knowledge Base Client
4132
4135
Knowledge Base Server
4139
Connection Manager
Data Base Manager
Dynamic Class Manager
Query Manager
Units Manager
4138
4136
Lockholder Table
4134
Handle Manager
4137
Legacy Manager
4145
4146
Lock Manager
4125
File Manager
4140
4123

FIG. 205

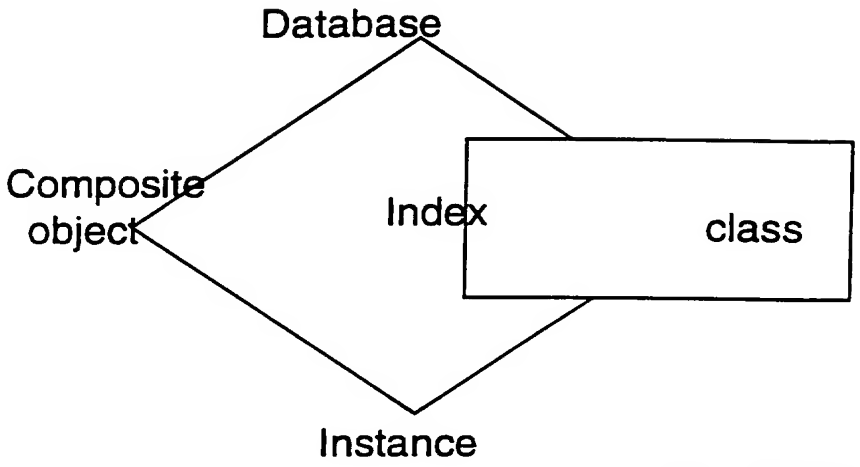


Fig. 206A

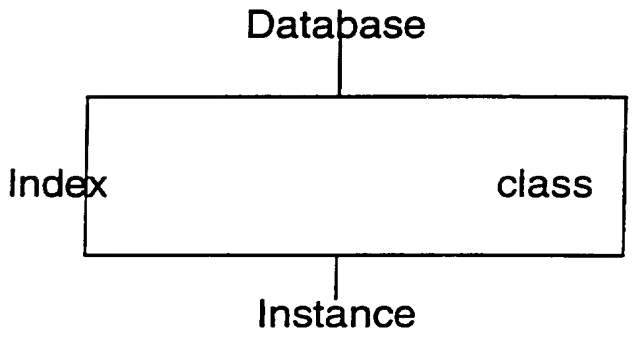


Fig. 206B

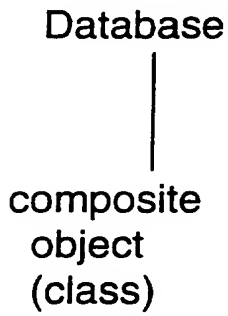


Fig. 206C

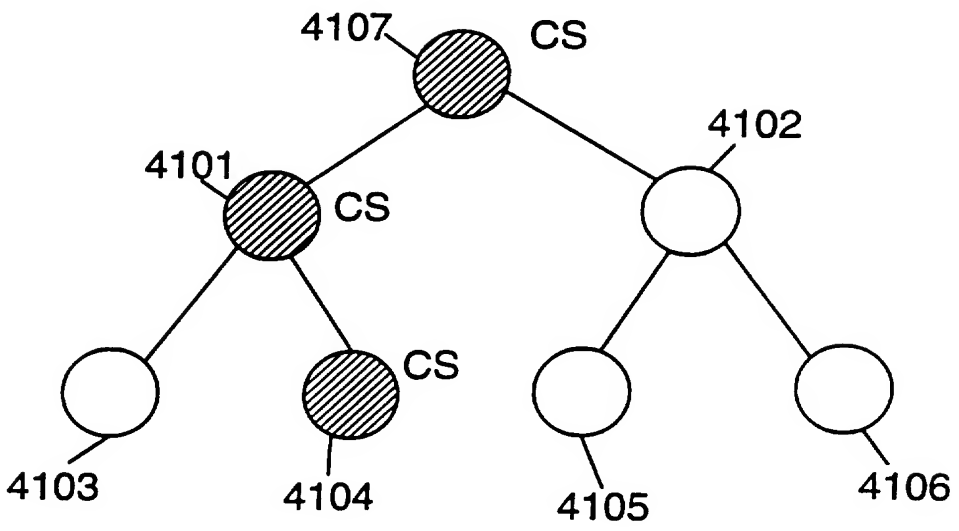


FIG. 207A

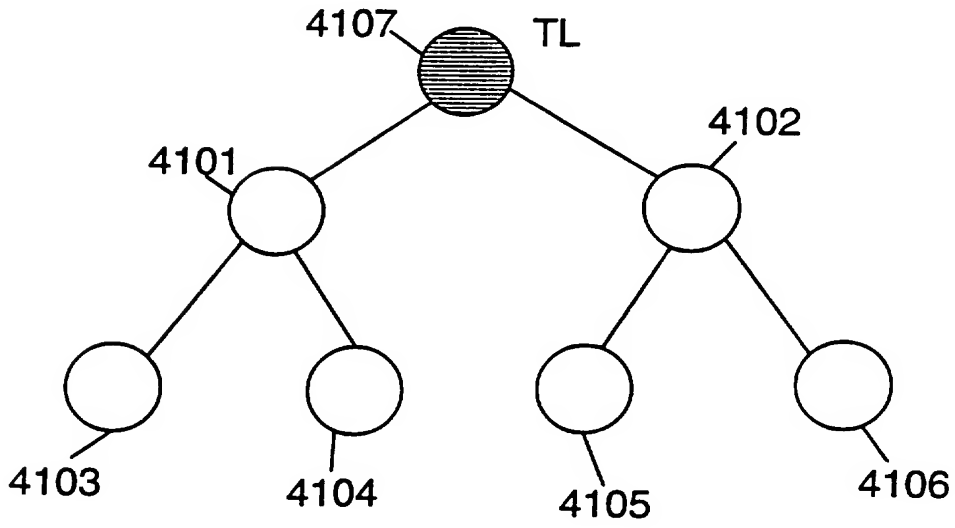
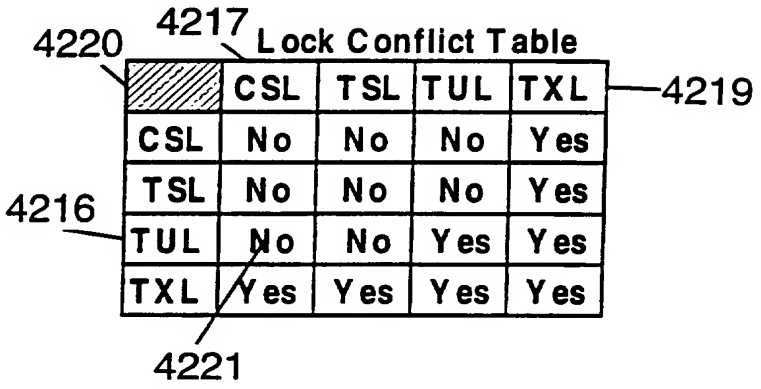


FIG. 207B



	CSL	TSL	TUL	TXL
CSL	No	No	No	Yes
TSL	No	No	No	Yes
TUL	No	No	Yes	Yes
TXL	Yes	Yes	Yes	Yes

FIG. 208

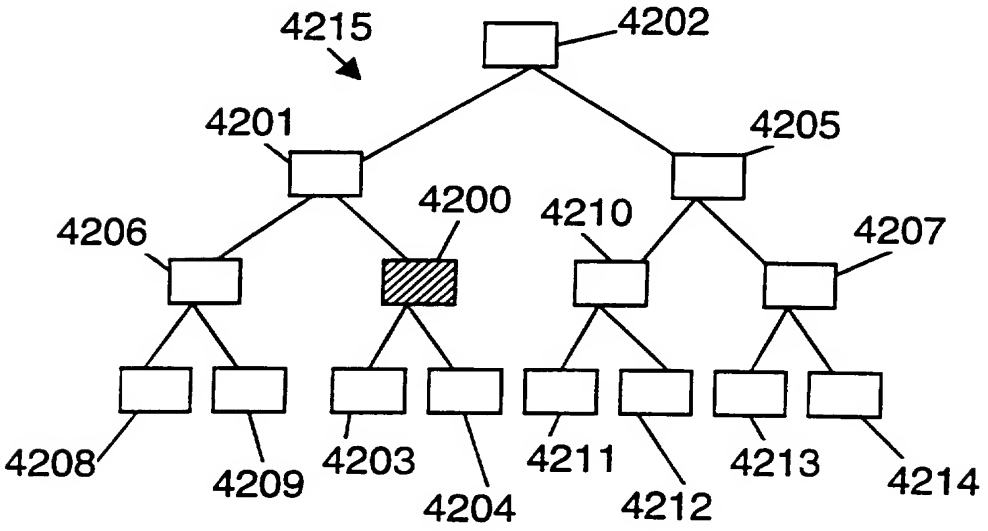


FIG. 209

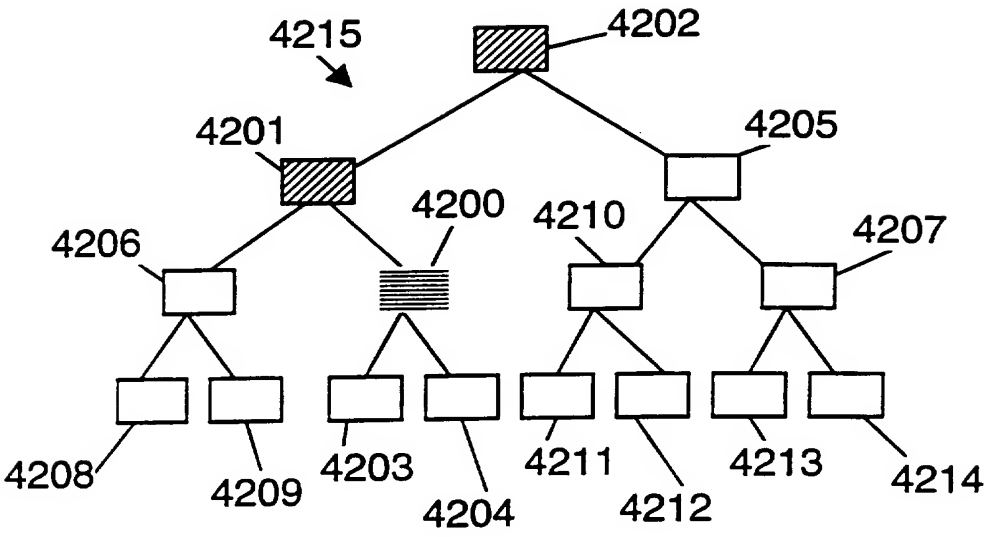


FIG. 210

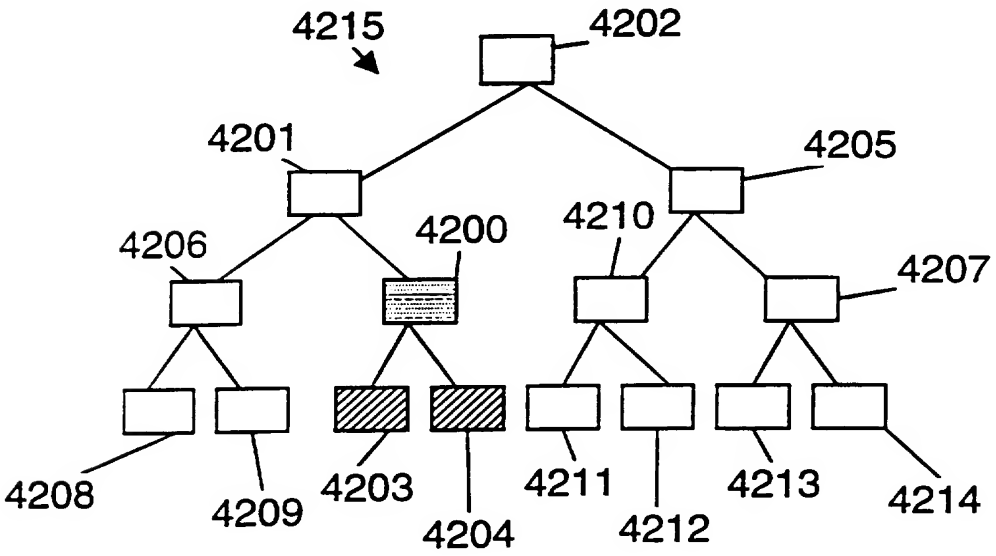


FIG. 211

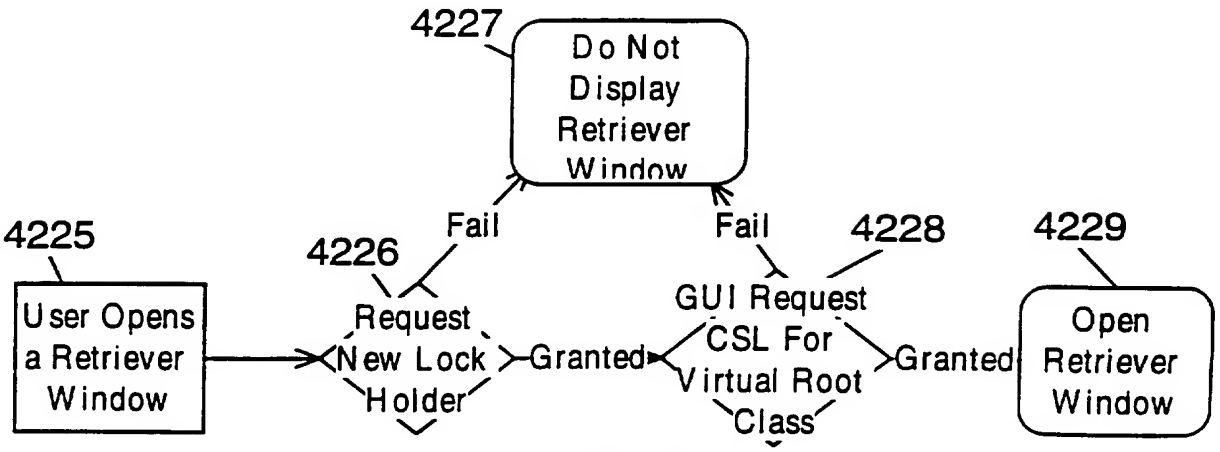


FIG. 212

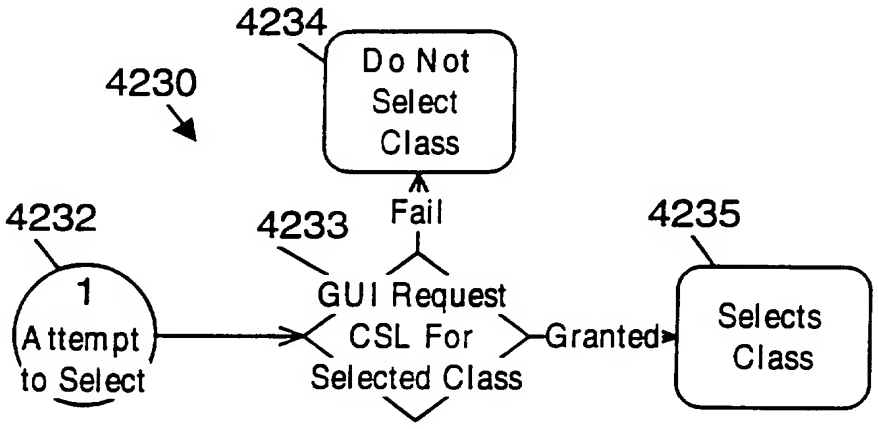


FIG. 213

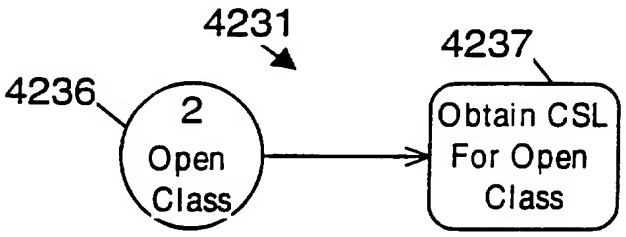


FIG. 214

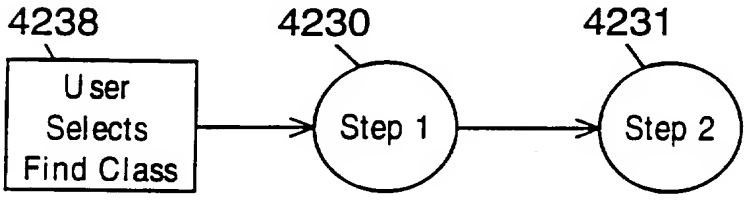


FIG. 215

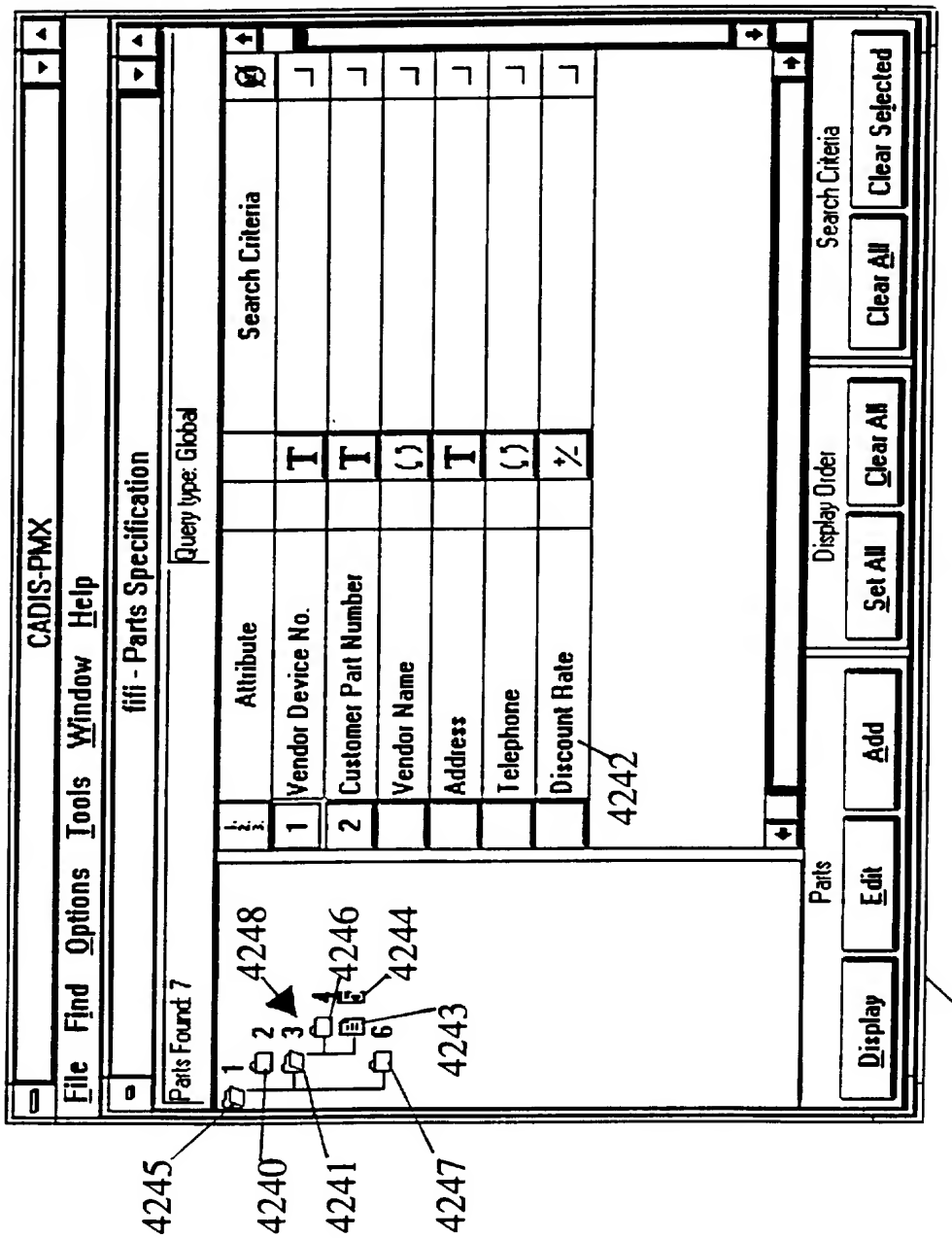

CADIS-PMX
File Find Options Tools Window Help
fifi - Parts Specification
Parts Found: 7
Query type: Global
Search Criteria
Attribute
Vendor Device No.
Customer Part Number
Vendor Name
Address
Telephone
Discount Rate
Search Criteria
Clear All
Clear Selected
Display Order
Set All
Clear All
Parts
Display
Edit
Add
4242
4248
4246
4244
4243
4245
4240
4241
4247
4290

FIG. 216

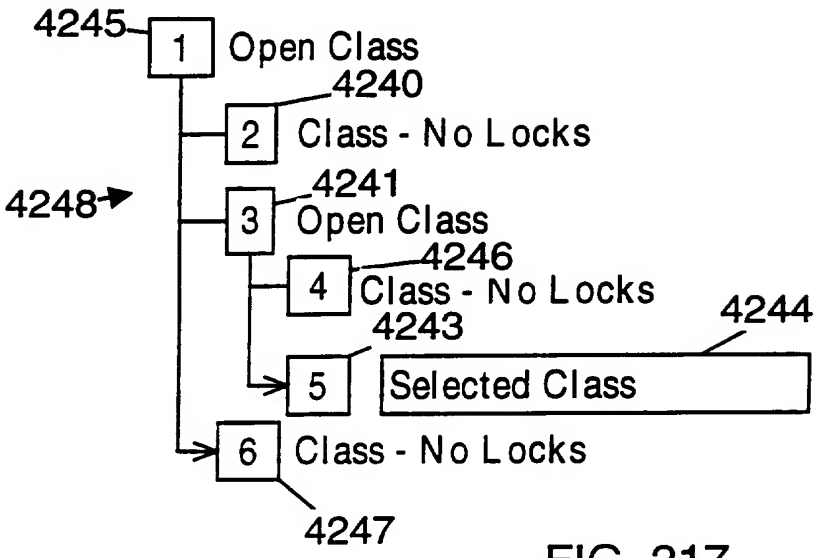


FIG. 217

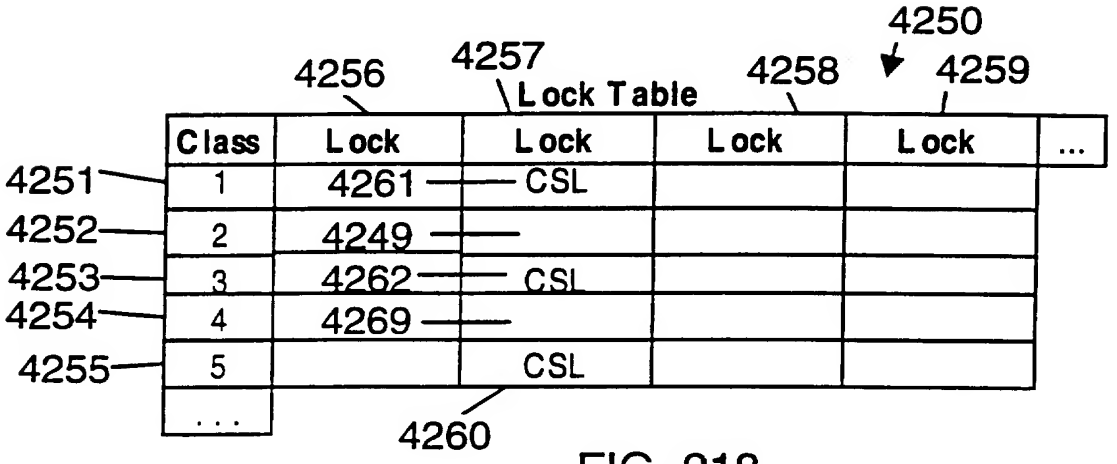


FIG. 218

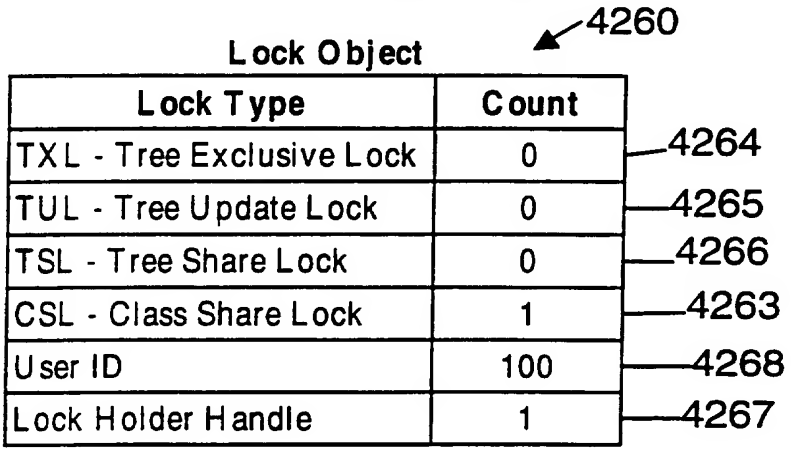


FIG. 219

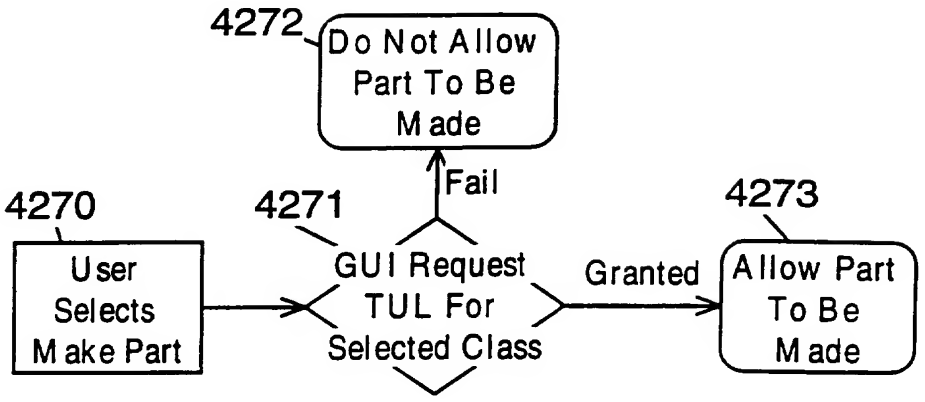


FIG. 220

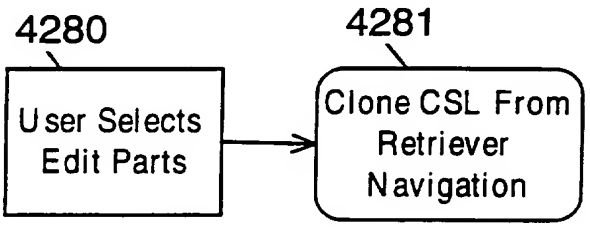


FIG. 225

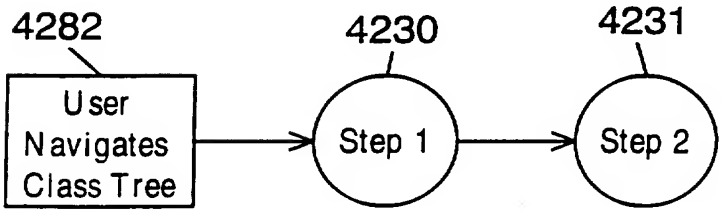


FIG. 226

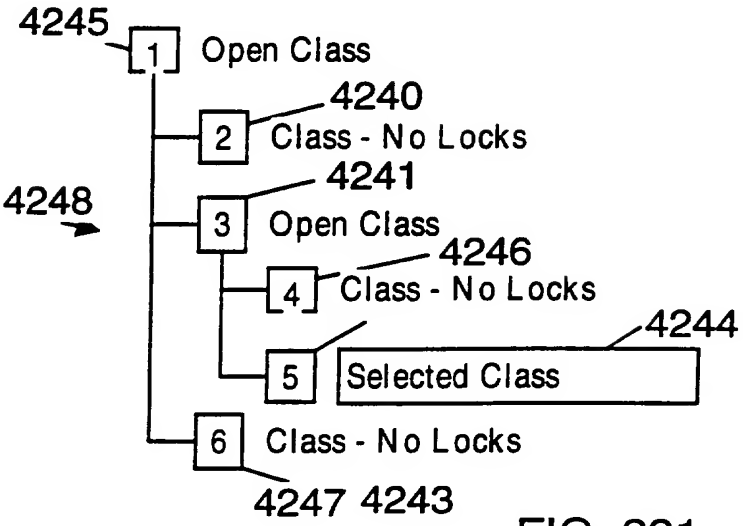


FIG. 221

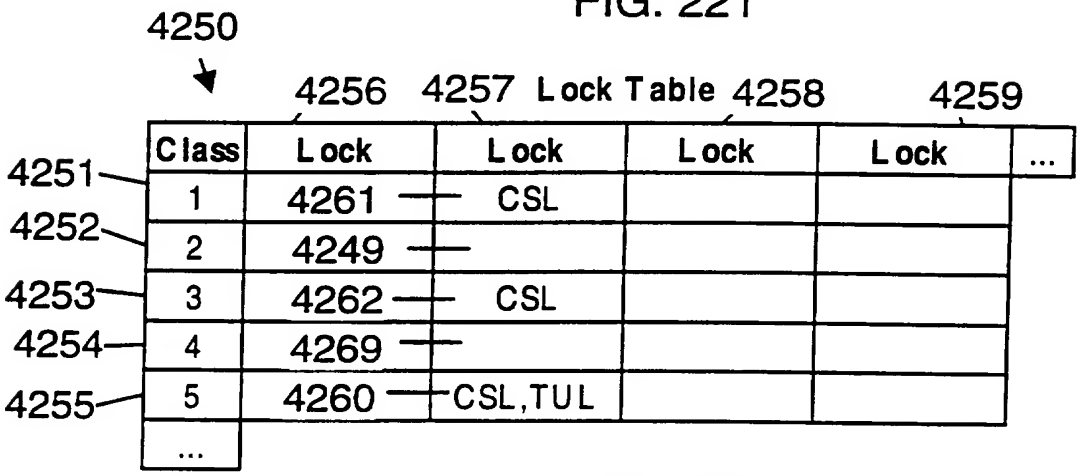


FIG. 222

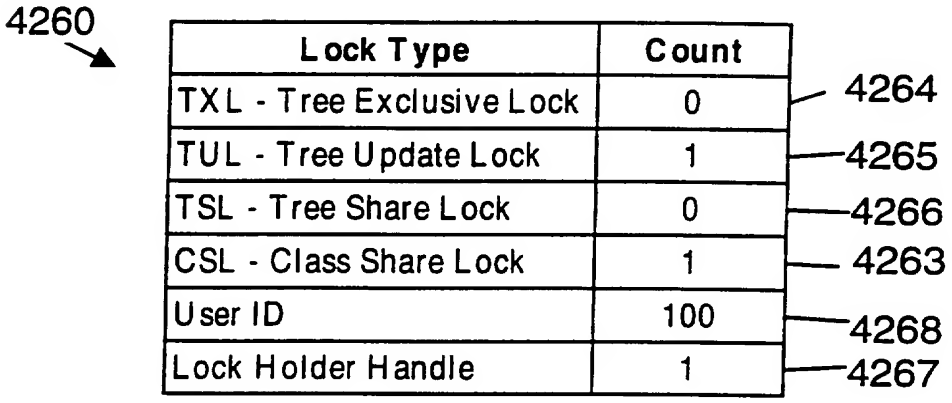


FIG. 223

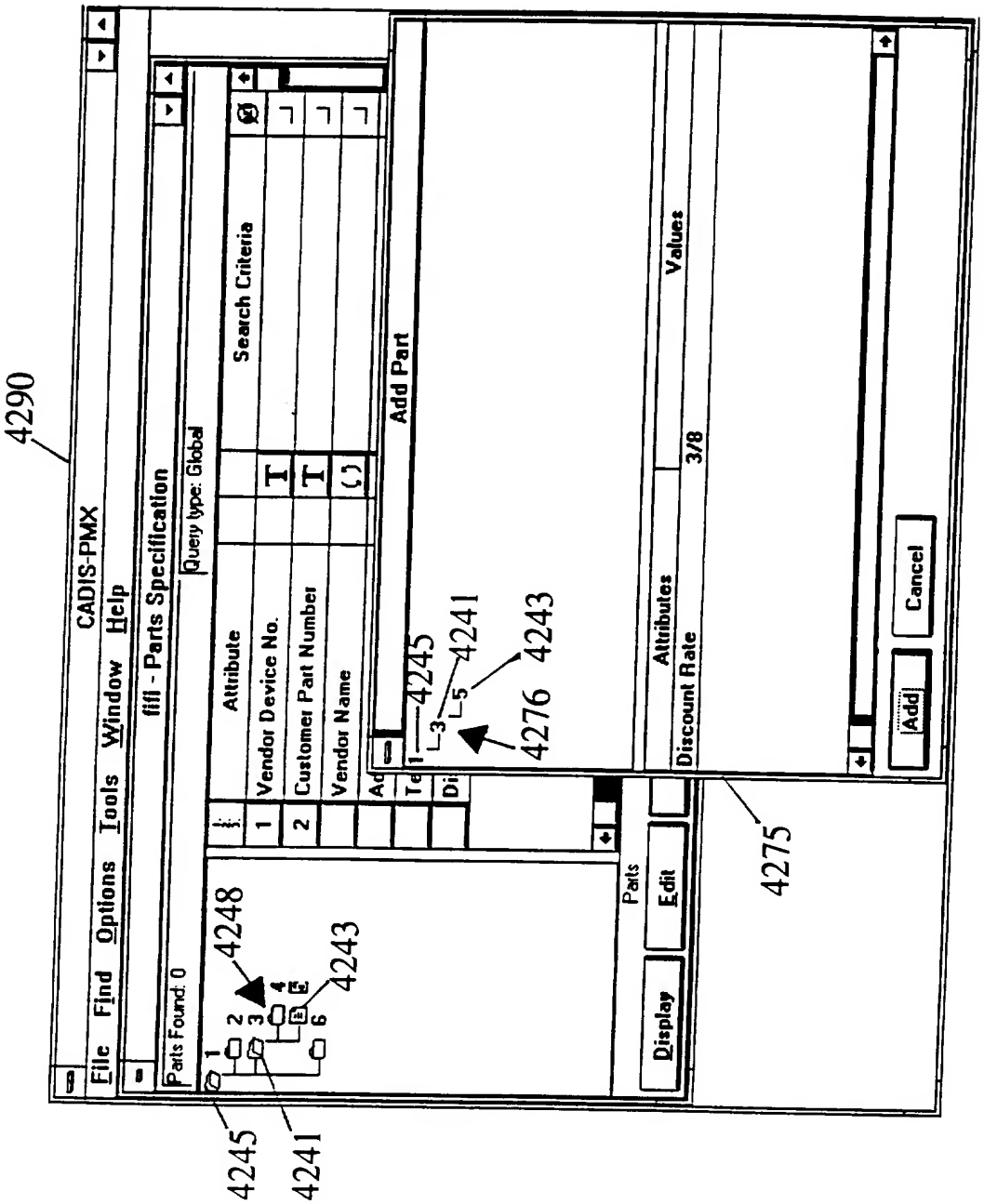

4290
CADIS-PMX
File Find Options Tools Window Help
fifi - Parts Specification
Query type: Global
Parts Found: 0
Search Criteria
Attribute
1 Vendor Device No.
2 Customer Part Number
Vendor Name
Add Part
Values
Attributes
3/8
Discount Rate
Parts
Display
Edit
Add
Cancel
4248
4243
4245
4241
4276
4275

FIG. 224

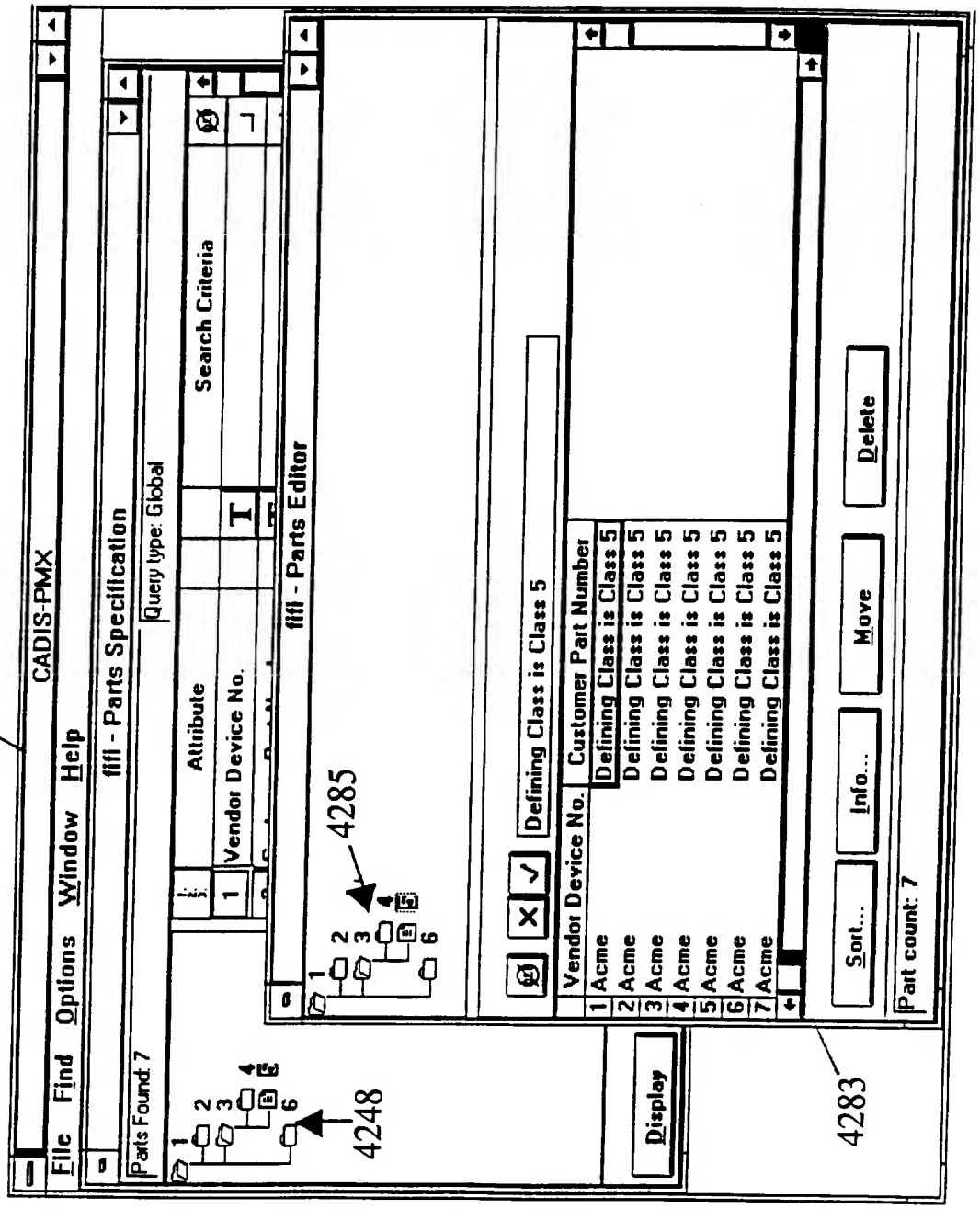

4290
CADIS-PMX
File Find Options Window Help
fifi - Parts Specification
Query type: Global
Parts Found: 7
Attribute
Search Criteria
Vendor Device No.
fifi - Parts Editor
4285
Defining Class is Class 5
Vendor Device No.
Customer Part Number
1 Acme Defining Class is Class 5
2 Acme Defining Class is Class 5
3 Acme Defining Class is Class 5
4 Acme Defining Class is Class 5
5 Acme Defining Class is Class 5
6 Acme Defining Class is Class 5
7 Acme Defining Class is Class 5
Sort...
Info...
Move
Delete
Part count: 7
Display
4248
4283

FIG. 227

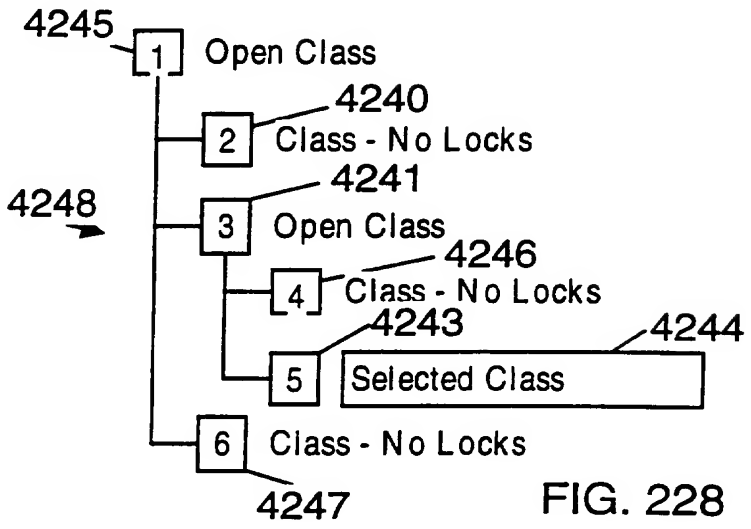


FIG. 228

4250

4256 4257 Lock Table 4258 4259

Class	Lock	Lock	Lock	Lock	...
1	4261	CSL, CSL			
2	4249				
3	4262	CSL, CSL			
4	4269				
5	4260	CSL,CSL			
...					

4251 4252 4253 4254 4255

FIG. 229

4260

Lock Type	Count
TXL - Tree Exclusive Lock	0
TUL - Tree Update Lock	0
TSL - Tree Share Lock	0
CSL - Class Share Lock	2
User ID	100
Lock Holder Handle	1

4264 4265 4266 4263 4268 4267

FIG. 230

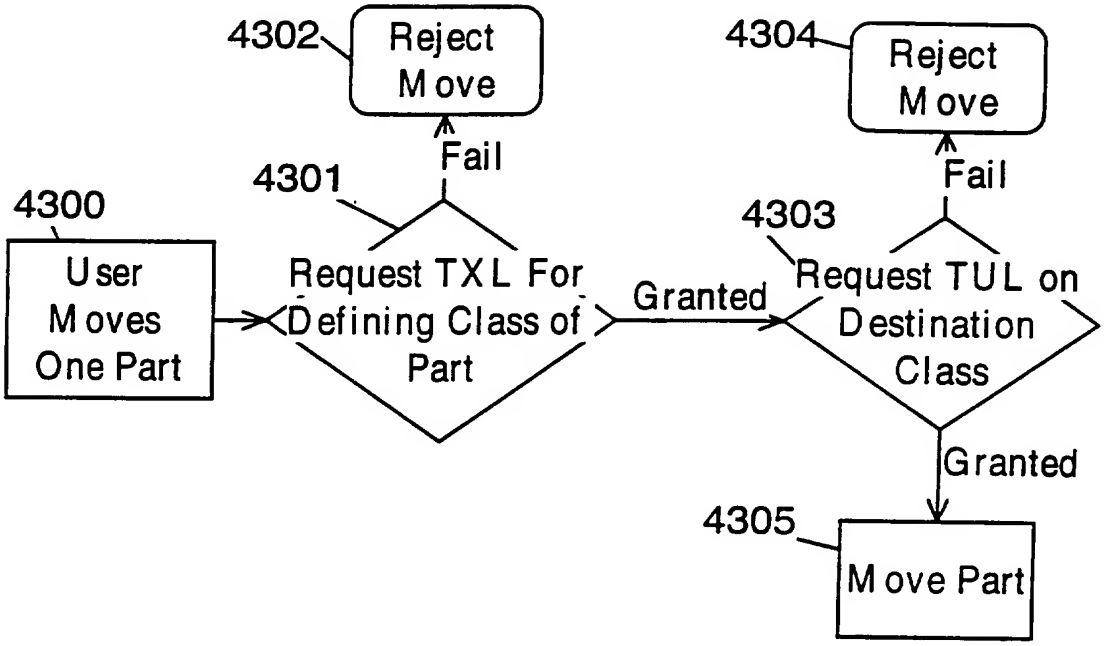


FIG. 231

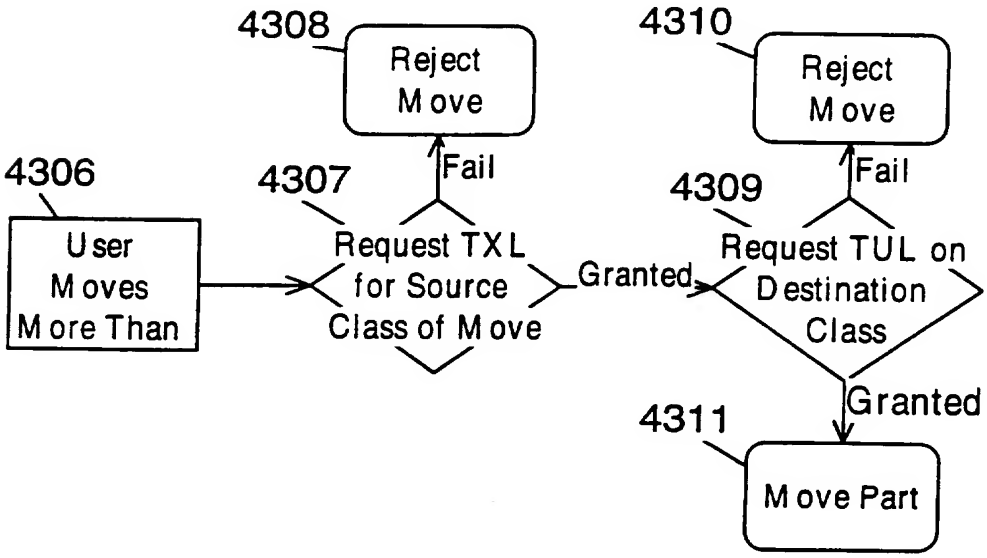


FIG. 232

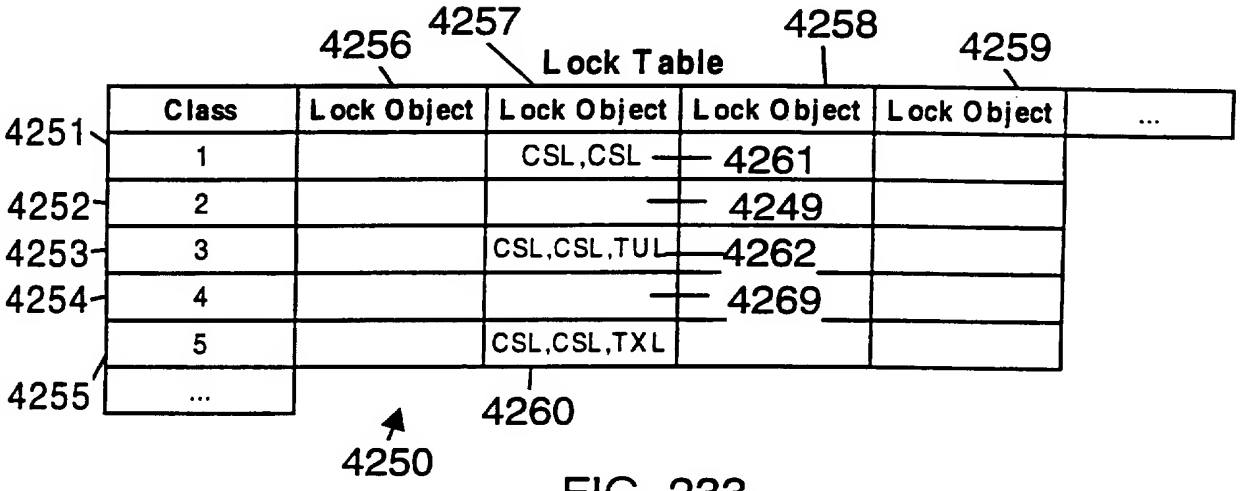


FIG. 233

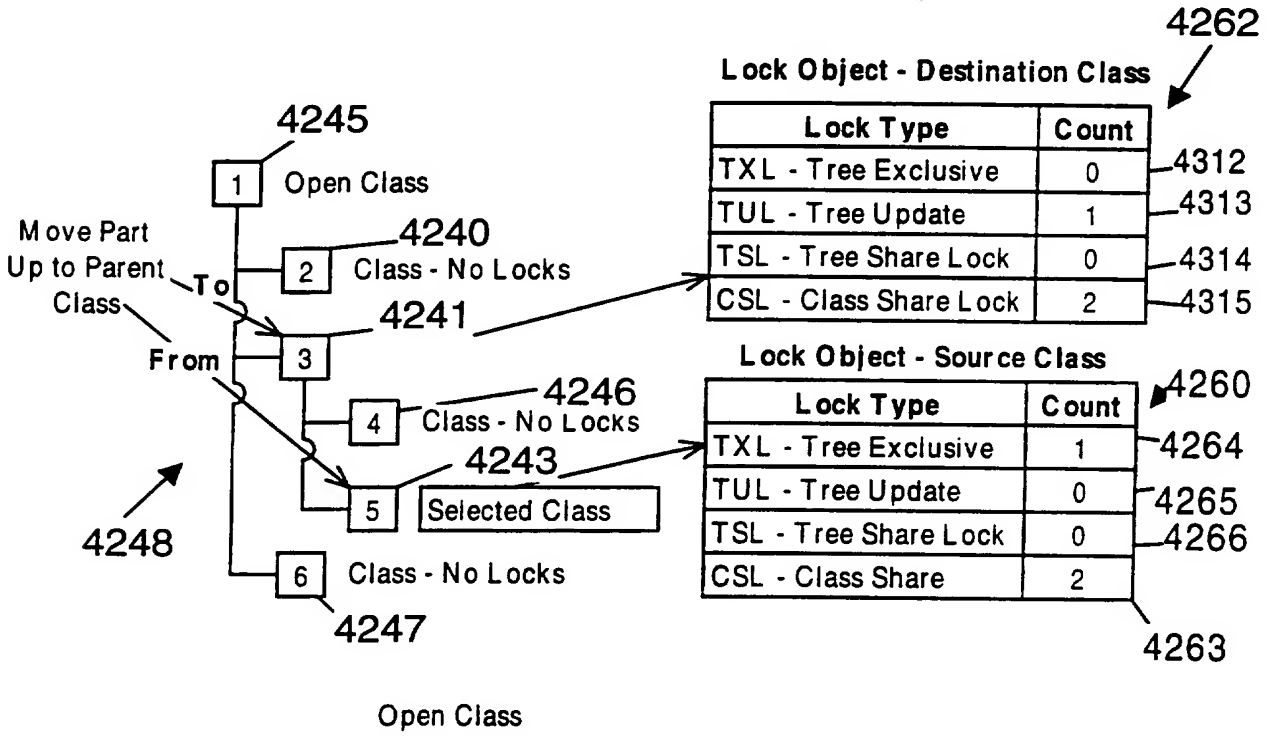


FIG. 234

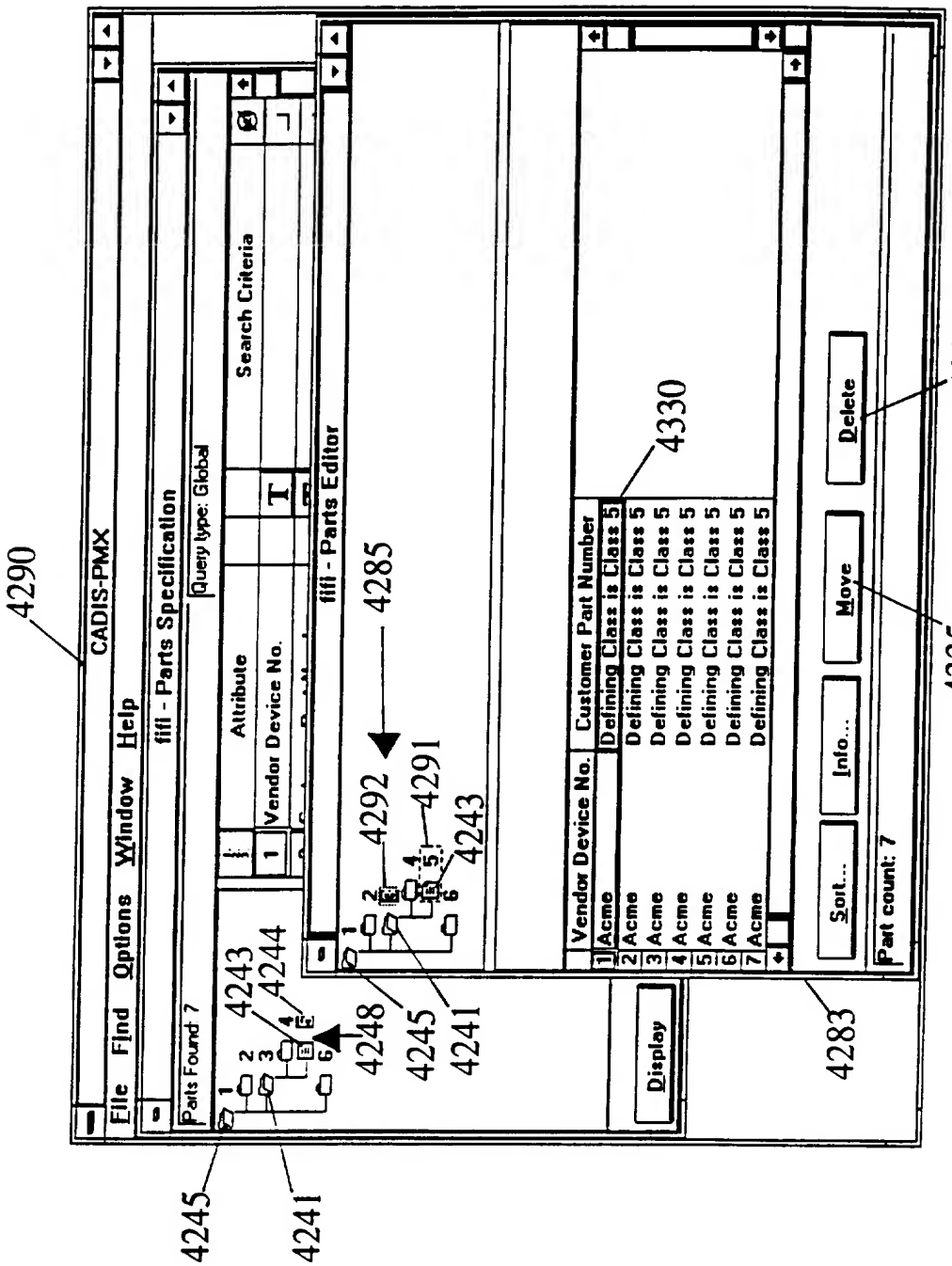

4290
CADIS-PMX
File Find Options Window Help
fifi - Parts Specification
Query type: Global
Parts Found: 7
Search Criteria
Attribute
Vendor Device No.
fifi - Parts Editor
4285
4292
4291
4243
4243
4244
4248
4245
4241
4245
4241
Vendor Device No.
Customer Part Number
Acme
Defining Class is Class 5
4330
Display
Sort...
Info...
Move
Delete
4331
4335
Part count: 7
4283

FIG. 235

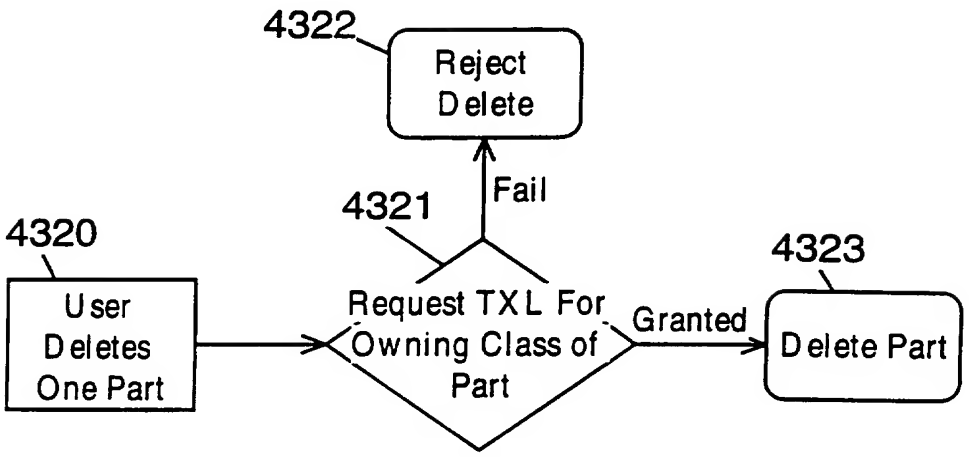


FIG. 236

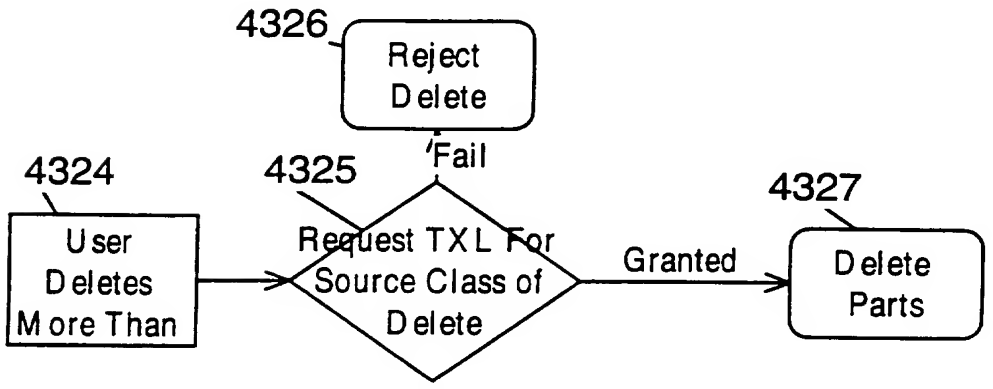


FIG. 237

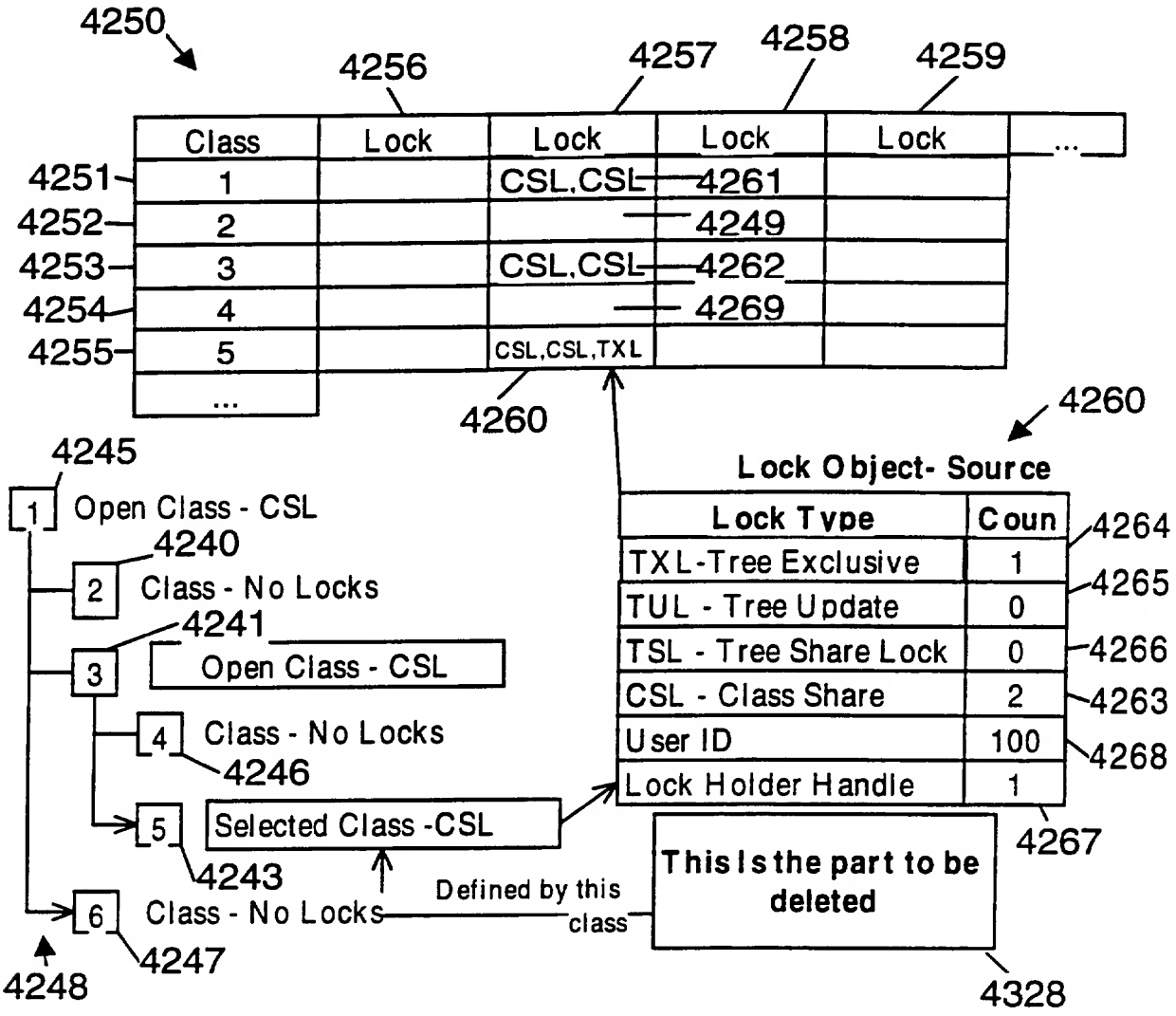

4250
4256
4257
4258
4259
Class
Lock
Lock
Lock
Lock
...
4251
1
CSL,CSL
4261
4252
2
4249
4253
3
CSL,CSL
4262
4254
4
4269
4255
5
CSL,CSL,TXL
...
4260
4260
Lock Object- Source
Lock Type
Coun
4264
TXL-Tree Exclusive
1
4265
TUL - Tree Update
0
TSL - Tree Share Lock
0
4266
CSL - Class Share
2
4263
User ID
100
4268
Lock Holder Handle
1
4267
4245
1
Open Class - CSL
4240
2
Class - No Locks
4241
3
Open Class - CSL
4
Class - No Locks
4246
5
Selected Class -CSL
4243
Defined by this class
This Is the part to be deleted
6
Class - No Locks
4247
4248
4328

FIG. 238

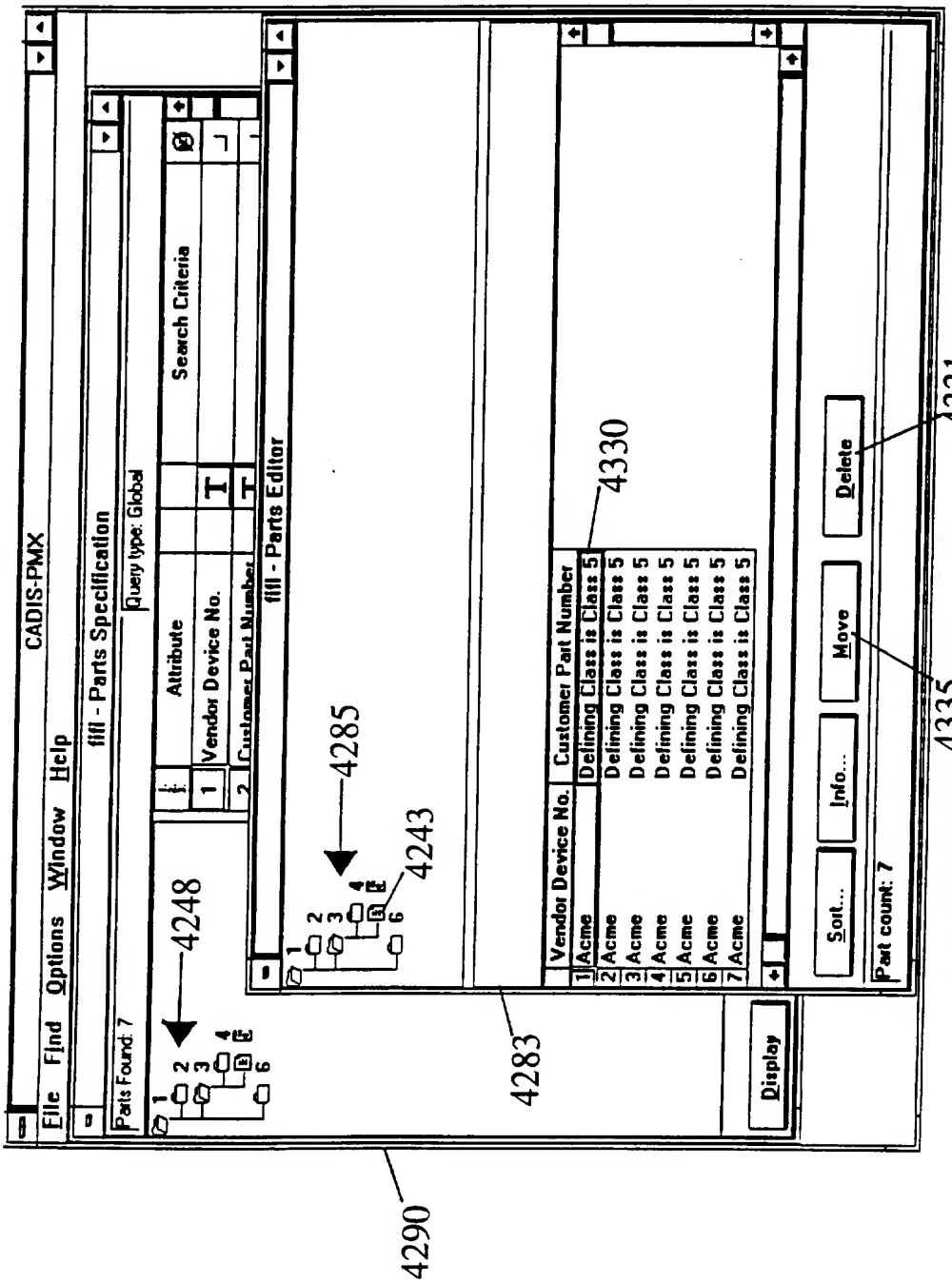

CADIS-PMX
File Find Options Window Help
fill - Parts Specification
Query type: Global
Parts Found: 7
Attribute
Search Criteria
1 Vendor Device No.
2 Customer Part Number
fill - Parts Editor
Vendor Device No.
Customer Part Number
1 Acme Defining Class is Class 5
2 Acme Defining Class is Class 5
3 Acme Defining Class is Class 5
4 Acme Defining Class is Class 5
5 Acme Defining Class is Class 5
6 Acme Defining Class is Class 5
7 Acme Defining Class is Class 5
Sort...
Info...
Move
Delete
Part count: 7
Display
4248
4285
4243
4330
4331
4335
4283
4290

FIG. 239

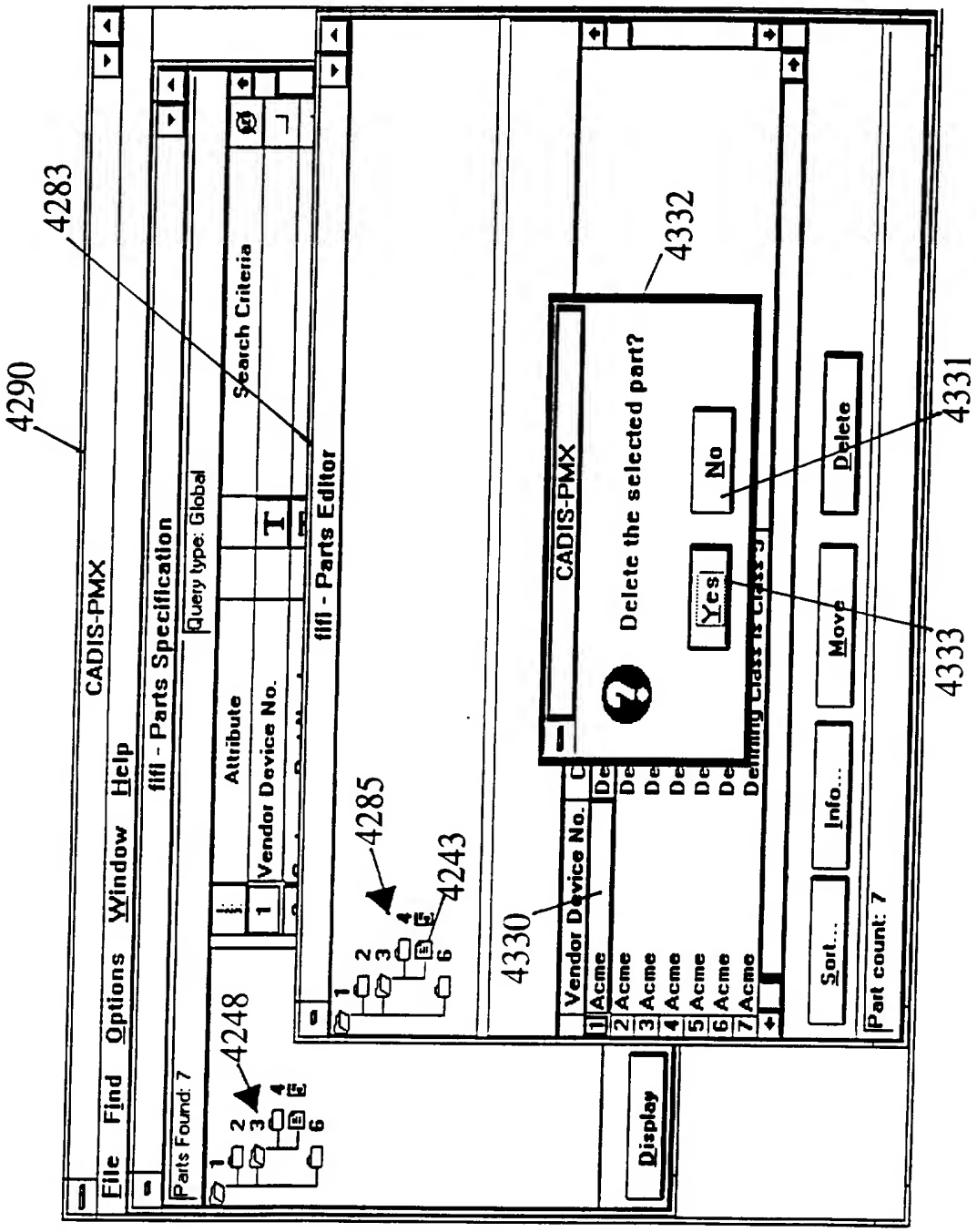

4290
4283
CADIS-PMX
File Find Options Window Help
fifi - Parts Specification
Query type: Global
Parts Found: 7
Search Criteria
Attribute
Vendor Device No.
fifi - Parts Editor
4248
4285
4243
4330
4332
4331
4333
CADIS-PMX
Delete the selected part?
Yes
No
Vendor Device No.
Acme
Display
Sort...
Info...
Move
Delete
Part count: 7

FIG. 240

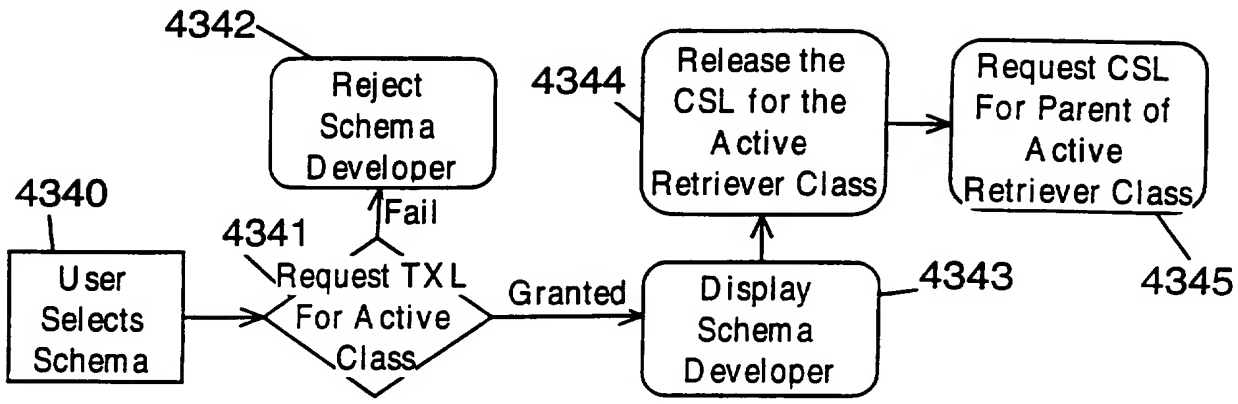


FIG. 241

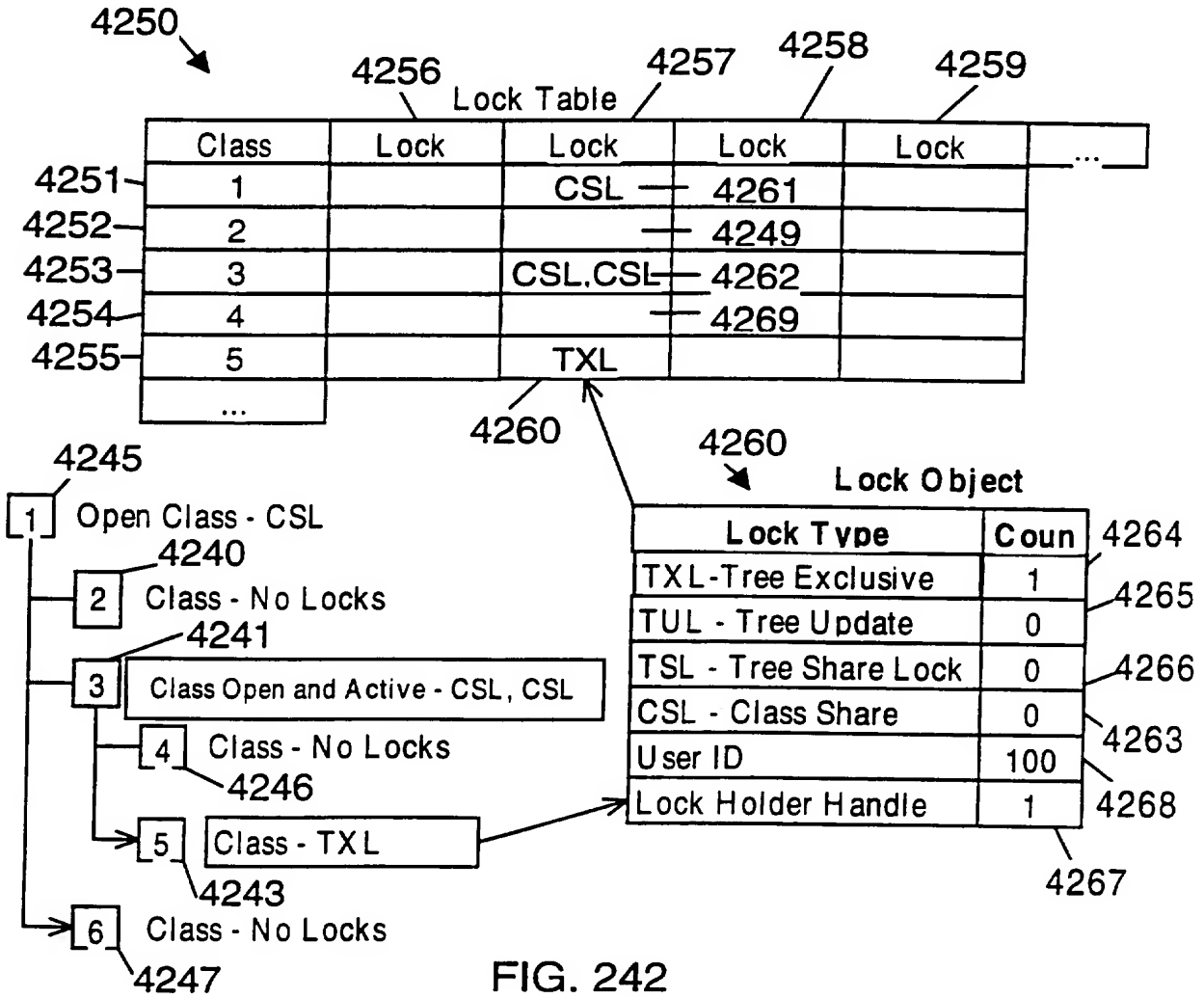


FIG. 242

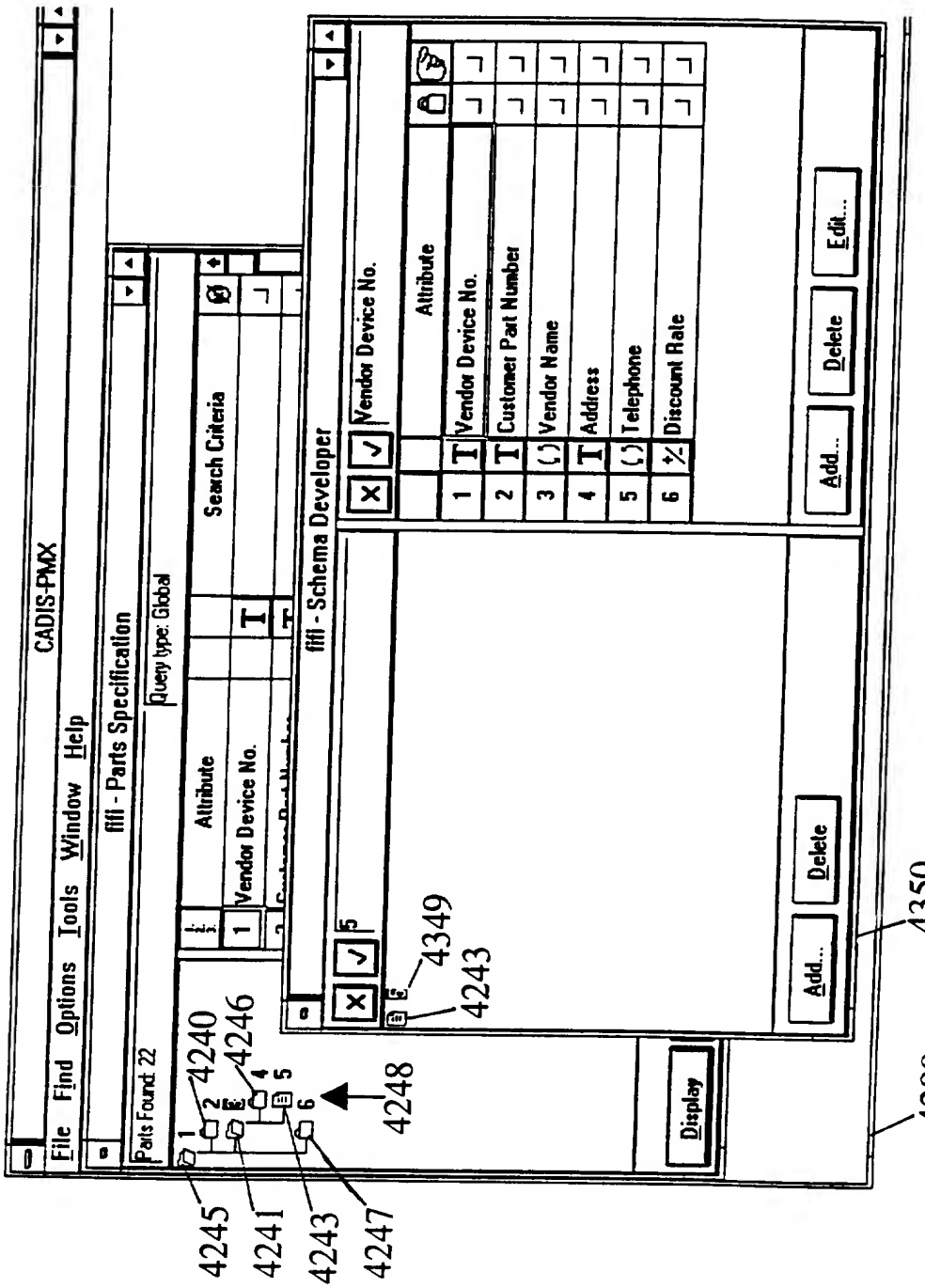

CADIS-PMX
File Find Options Tools Window Help
fifi - Parts Specification
Parts Found: 22
Query type: Global
Search Criteria
Attribute
Vendor Device No.
fifi - Schema Developer
Vendor Device No.
Attribute
Vendor Device No.
Customer Part Number
Vendor Name
Address
Telephone
Discount Rate
Add...
Delete
Edit...
Display
4240
4246
4248
4349
4243
4350
4290
4245
4241
4243
4247

FIG. 243

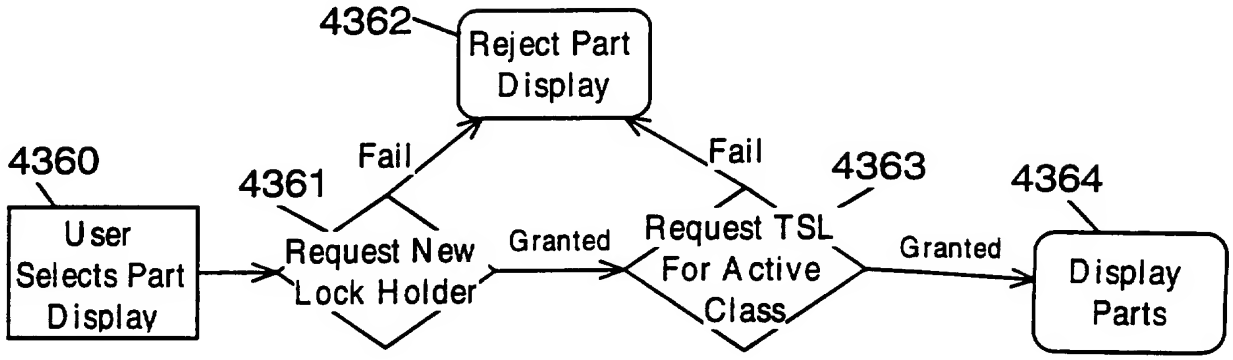


FIG. 244

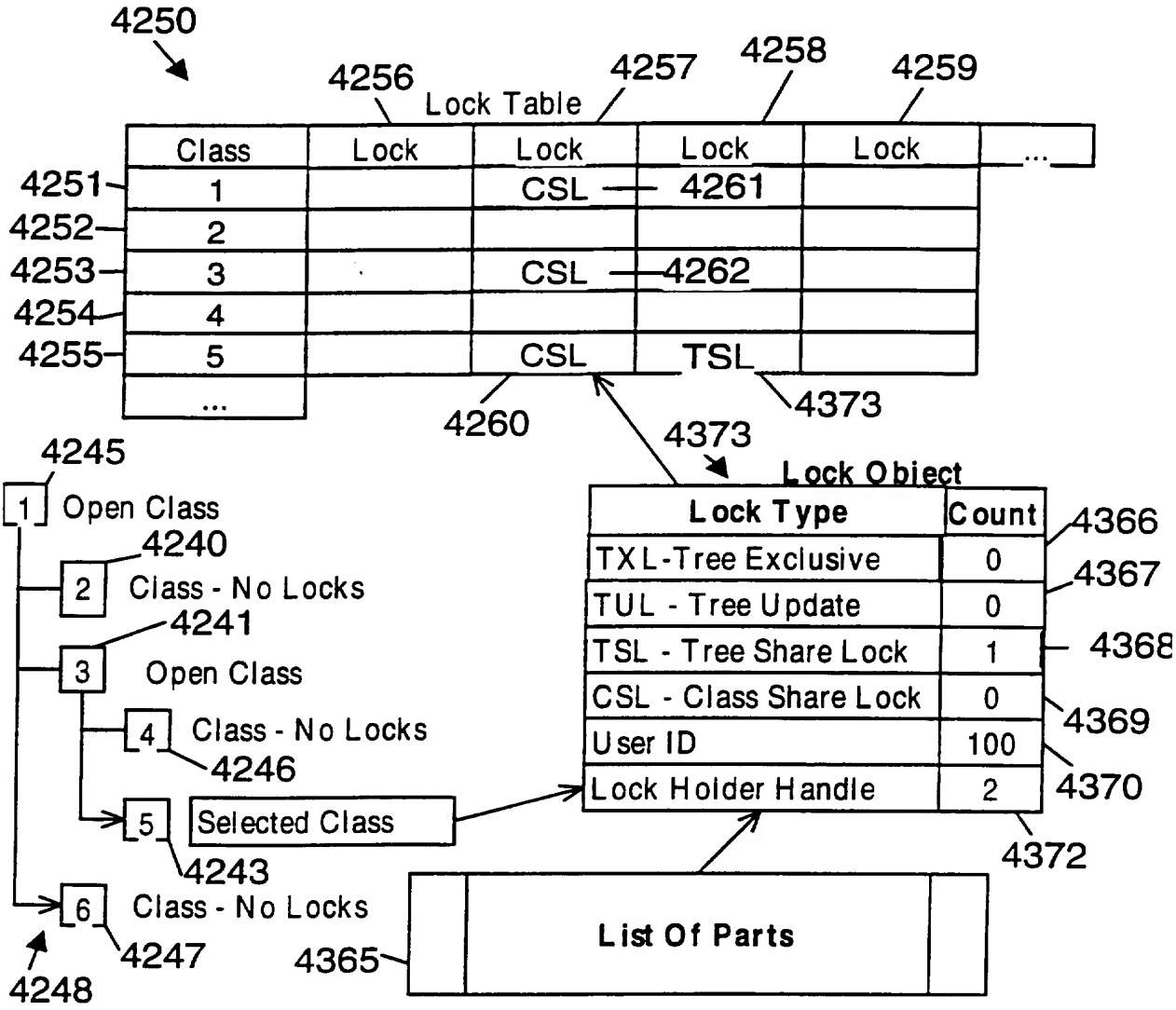


FIG. 245

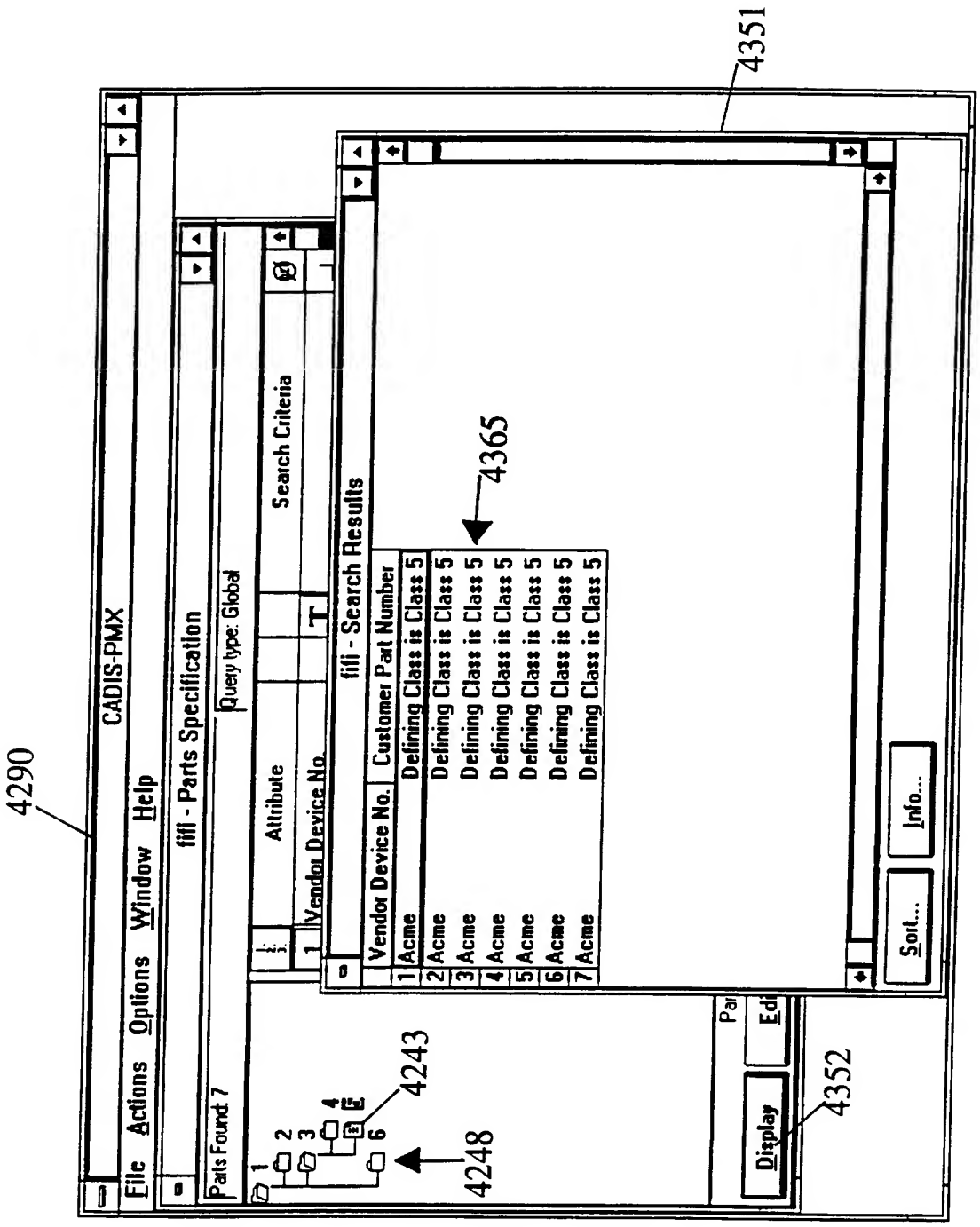

4290
4351
4365
4243
4248
4352
CADIS-PMX
File Actions Options Window Help
fifi - Parts Specification
Parts Found 7
Query type: Global
Attribute
Search Criteria
Vendor Device No.
fifi - Search Results
Vendor Device No.
Customer Part Number
Acme
Defining Class is Class 5
Sort...
Info...
Display

FIG. 246

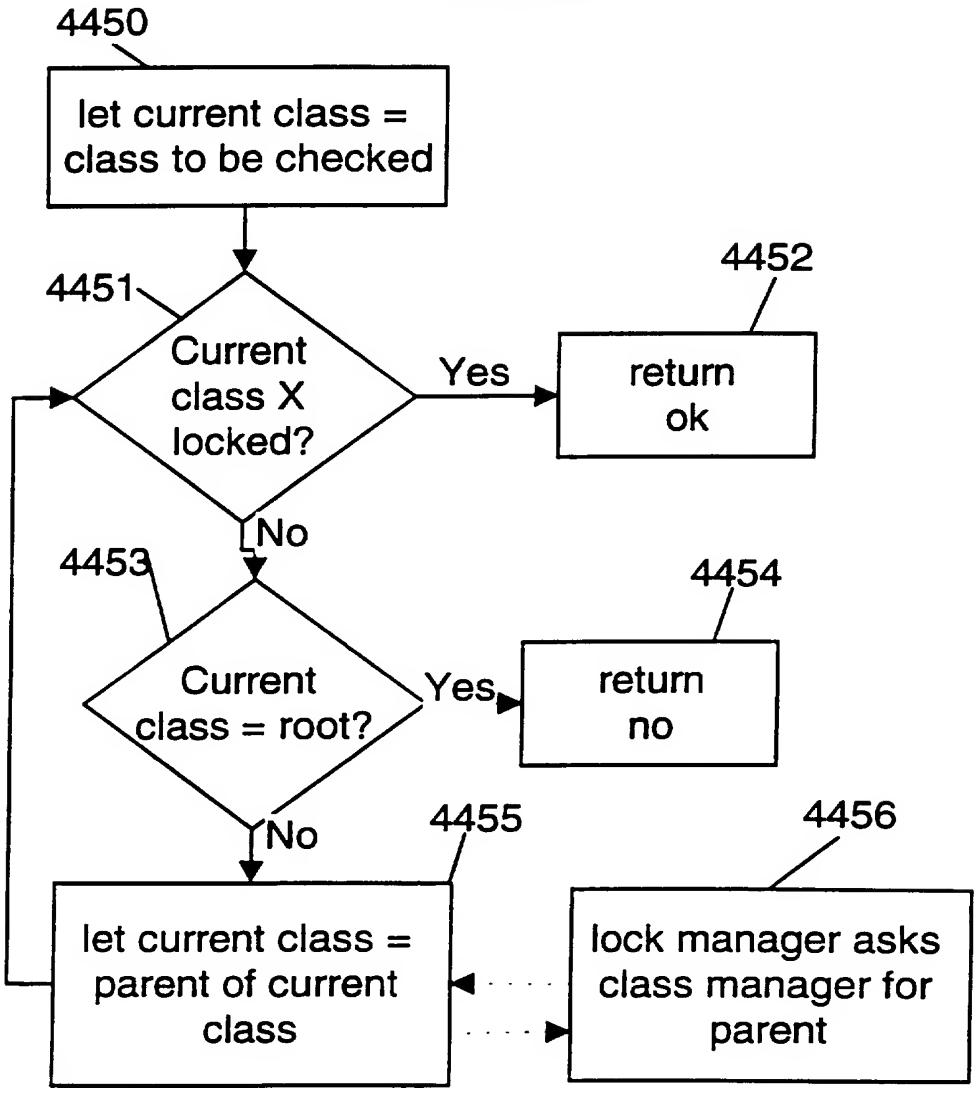

4450
let current class = class to be checked
4451
Current class X locked?
Yes
4452
return ok
No
4453
Current class = root?
Yes
4454
return no
No
4455
let current class = parent of current class
4456
lock manager asks class manager for parent

FIG. 247

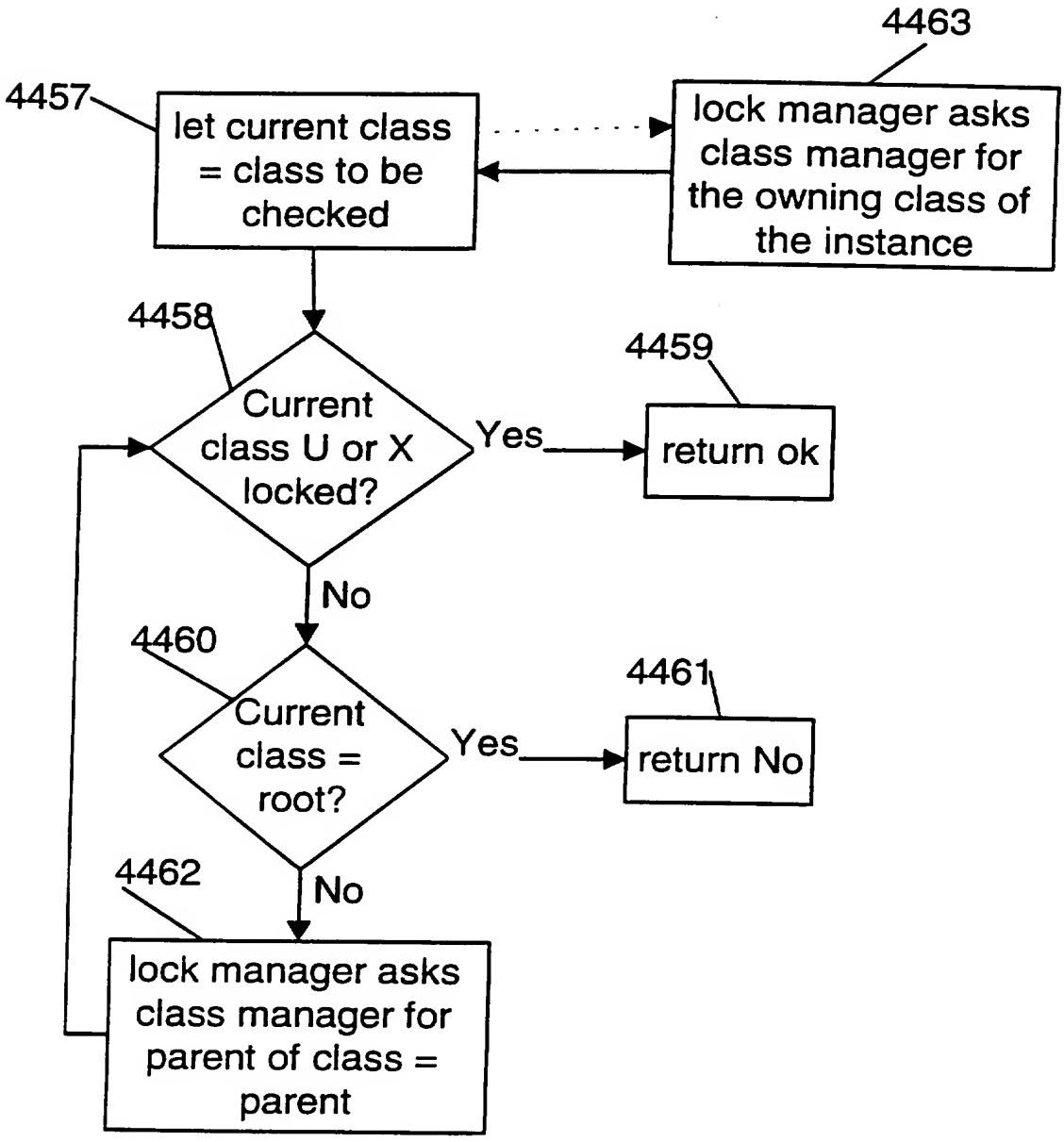

4463
4457
let current class = class to be checked
lock manager asks class manager for the owning class of the instance
4458
4459
Current class U or X locked?
Yes
return ok
No
4460
4461
Current class = root?
Yes
return No
4462
No
lock manager asks class manager for parent of class = parent

FIG. 248

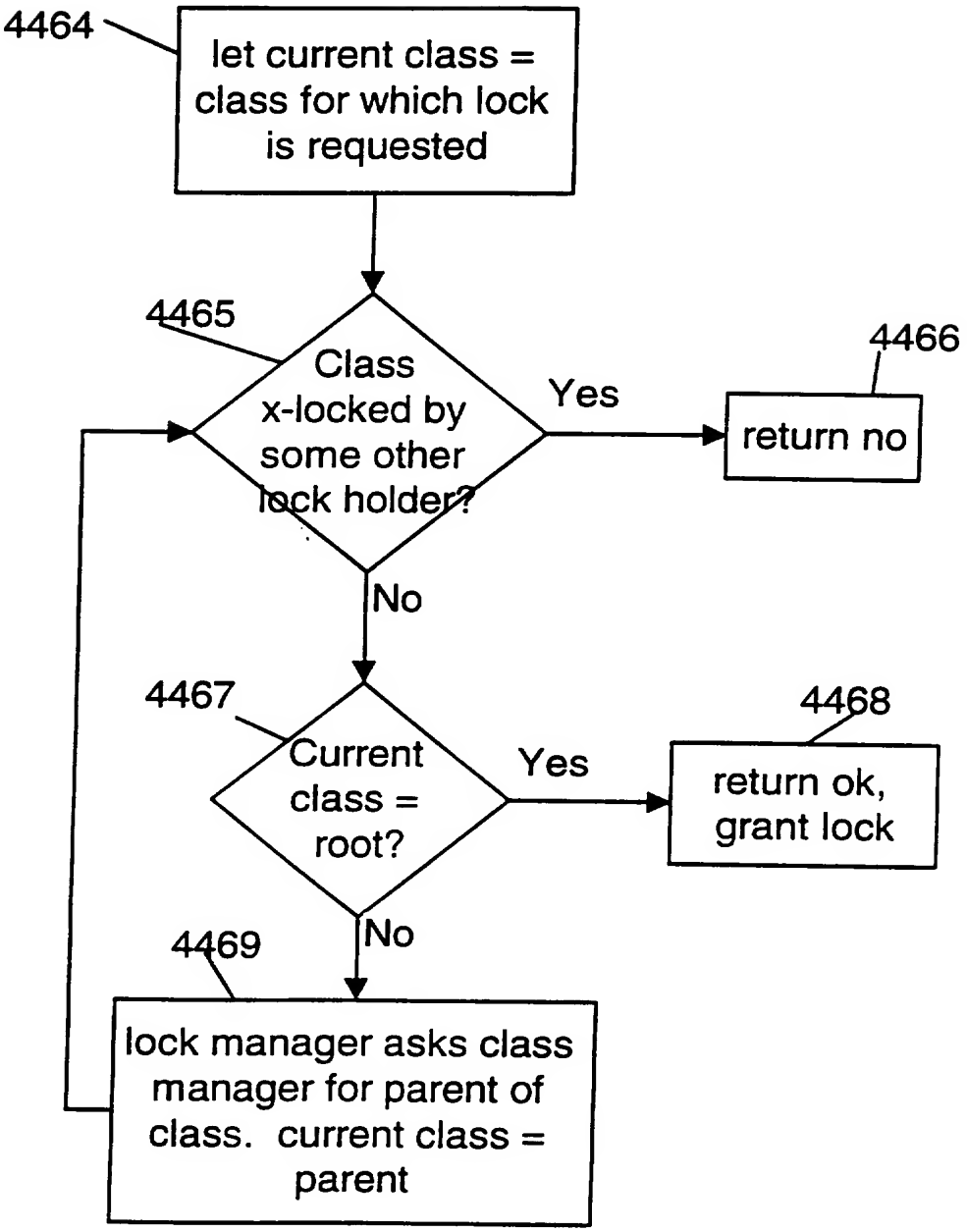

4464
let current class = class for which lock is requested
4465
Class x-locked by some other lock holder?
Yes
4466
return no
No
4467
Current class = root?
Yes
4468
return ok, grant lock
No
4469
lock manager asks class manager for parent of class. current class = parent

FIG. 249

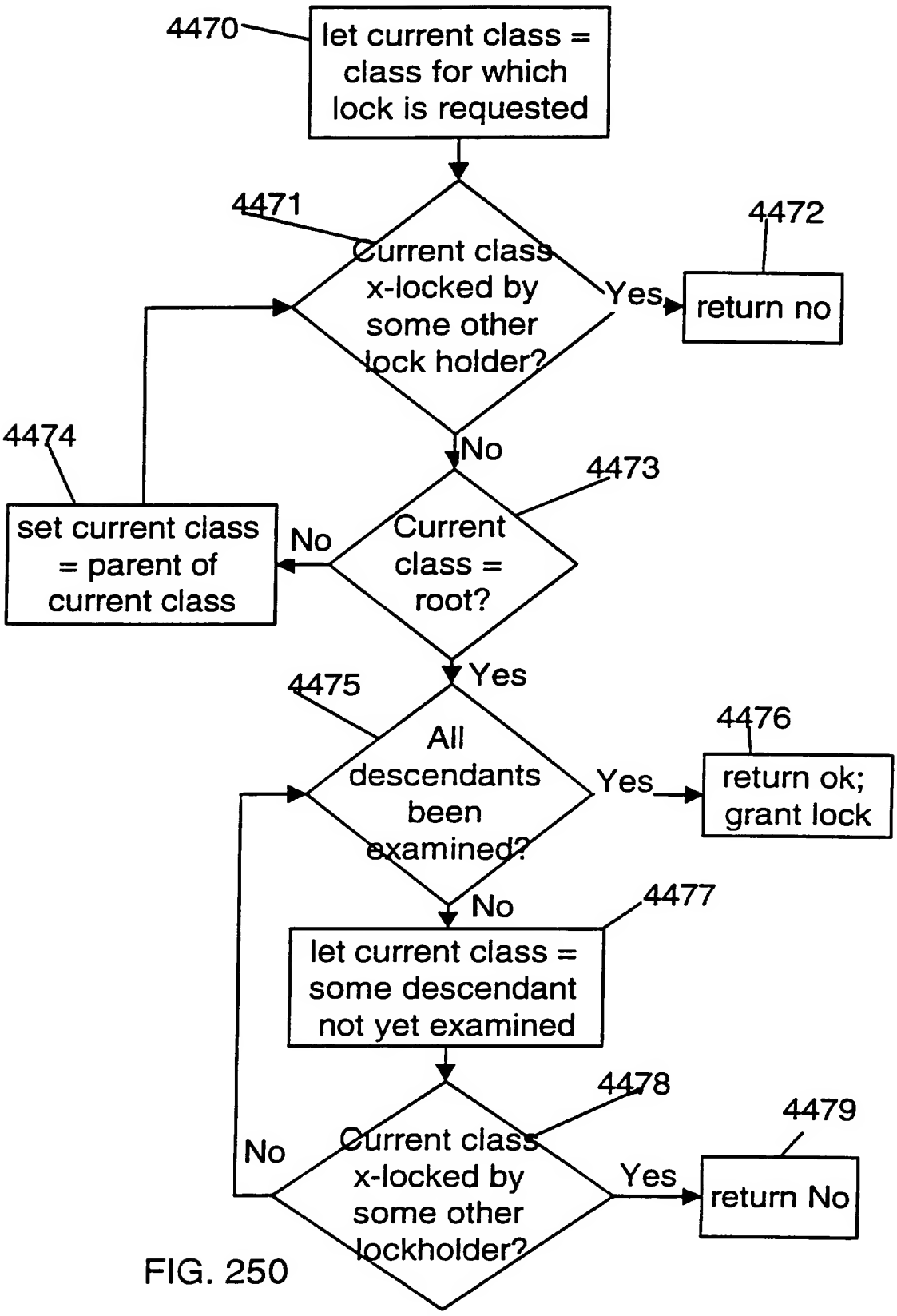

4470
let current class = class for which lock is requested
4471
Current class x-locked by some other lock holder?
Yes
4472
return no
No
4473
Current class = root?
No
4474
set current class = parent of current class
Yes
4475
All descendants been examined?
Yes
4476
return ok; grant lock
No
4477
let current class = some descendant not yet examined
4478
Current class x-locked by some other lockholder?
No
Yes
4479
return No

FIG. 250

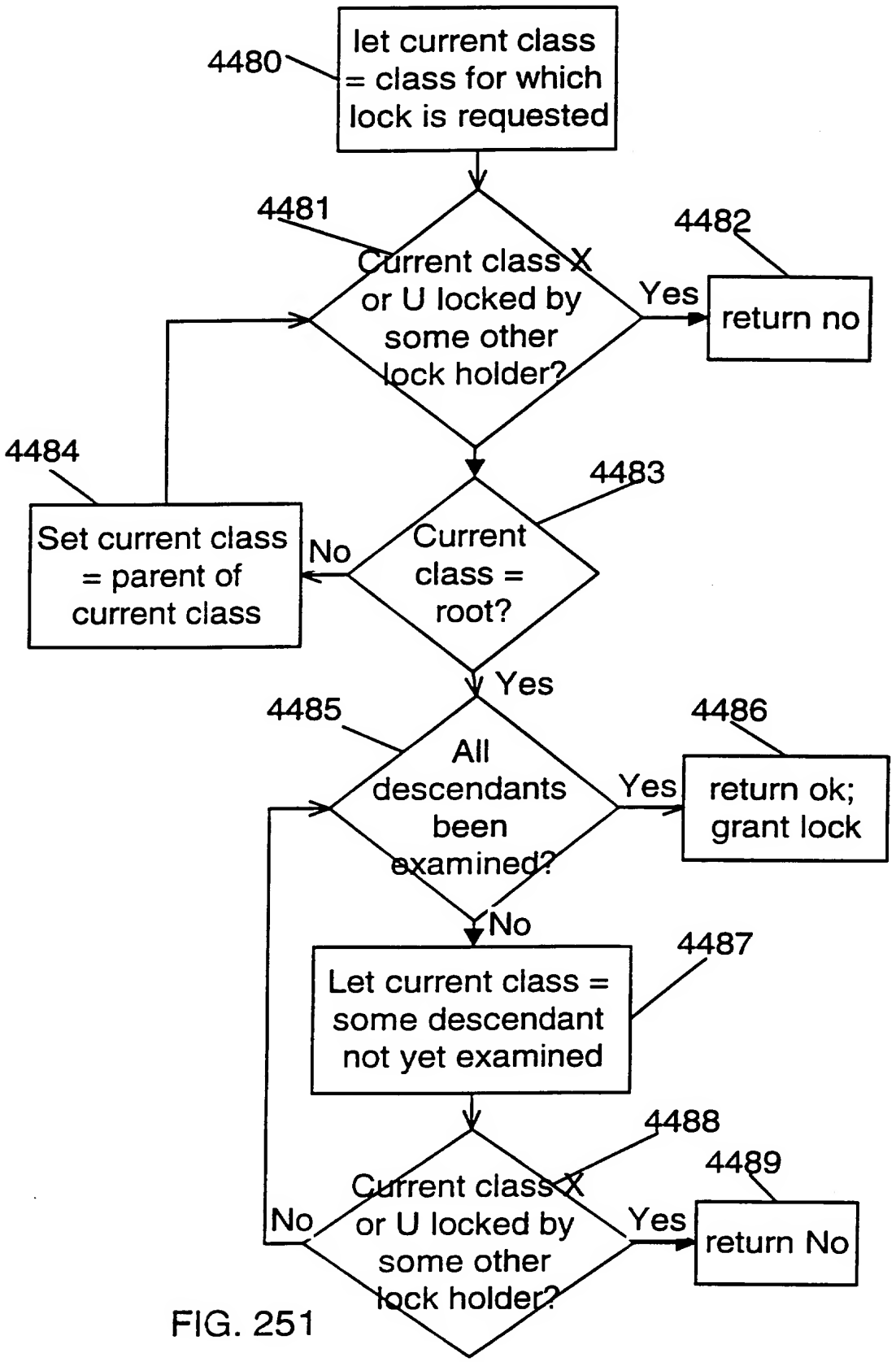

let current class = class for which lock is requested
4480
4481
Current class X or U locked by some other lock holder?
Yes
4482
return no
4484
Set current class = parent of current class
No
4483
Current class = root?
Yes
4485
All descendants been examined?
Yes
4486
return ok; grant lock
No
4487
Let current class = some descendant not yet examined
4488
Current class X or U locked by some other lock holder?
No
Yes
4489
return No

FIG. 251

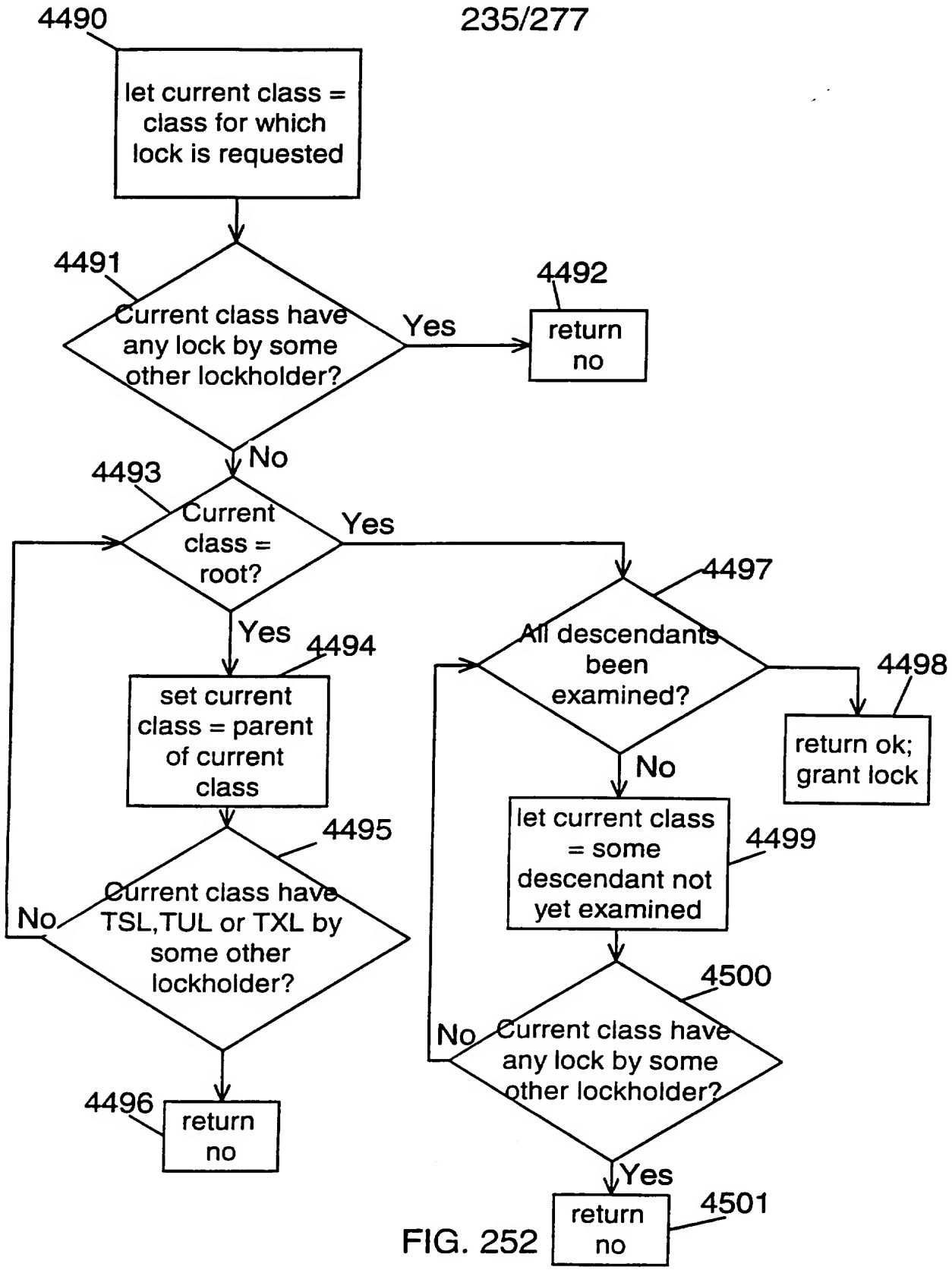

4490
let current class = class for which lock is requested
4491
Current class have any lock by some other lockholder?
Yes
4492
return no
No
4493
Current class = root?
Yes
Yes
4494
set current class = parent of current class
4495
Current class have TSL,TUL or TXL by some other lockholder?
No
4496
return no
4497
All descendants been examined?
4498
return ok; grant lock
No
let current class = some descendant not yet examined
4499
4500
Current class have any lock by some other lockholder?
No
Yes
return no
4501

FIG. 252

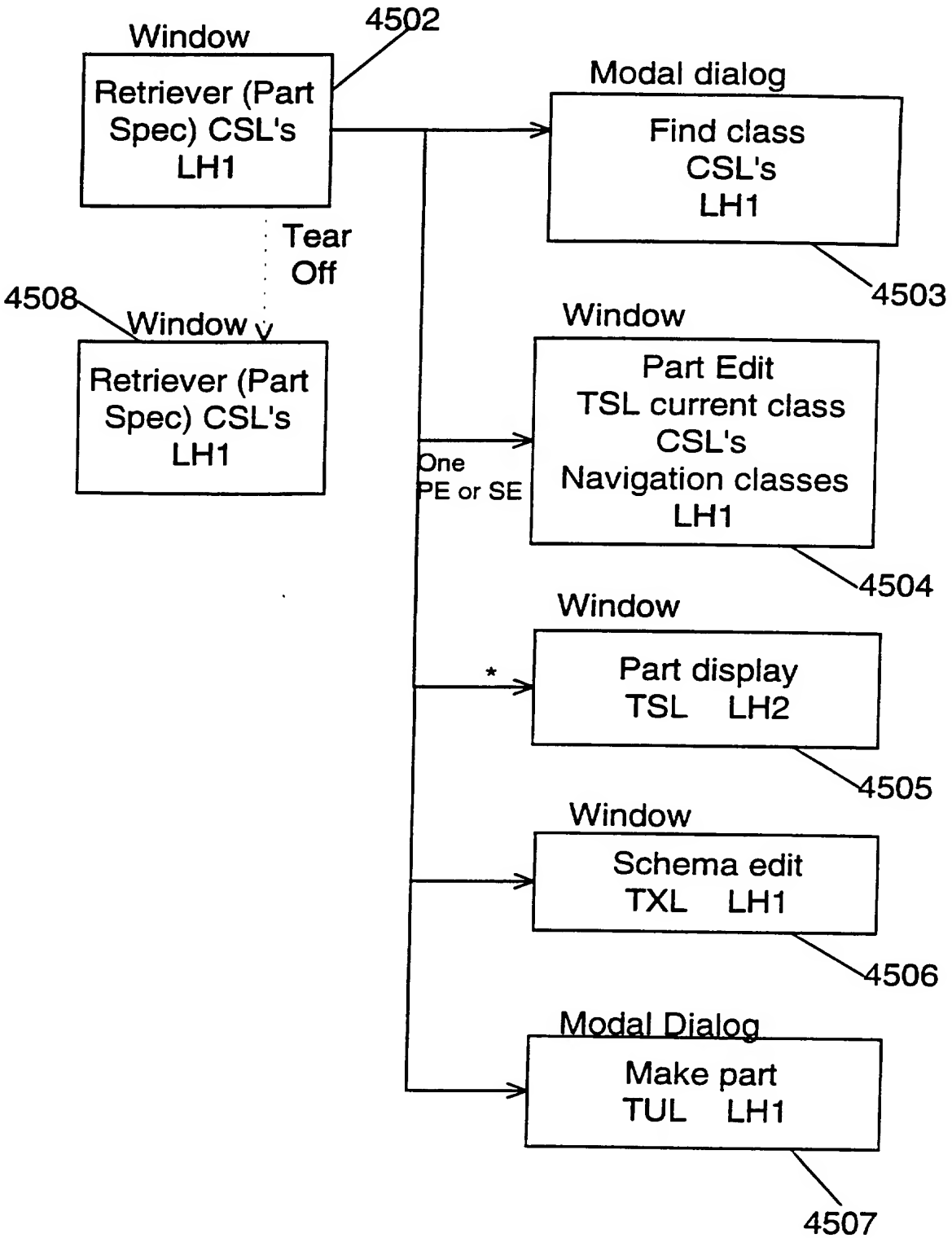

Window
Retriever (Part Spec) CSL's LH1
4502
Modal dialog
Find class CSL's LH1
4503
Tear Off
4508
Window
Retriever (Part Spec) CSL's LH1
Window
Part Edit TSL current class CSL's Navigation classes LH1
4504
One PE or SE
Window
Part display TSL LH2
4505
*
Window
Schema edit TXL LH1
4506
Modal Dialog
Make part TUL LH1
4507

FIG. 253

	LOCK HOLDER 1	LOCK HOLDER 2	LOCK HOLDER 3	LOCK HOLDER 4	LOCK HOLDER 5	LOCK HOLDER 6	LOCK HOLDER 7
CLASS HANDLE 1	4401	4408	4415	4422	4429	4436	4443
CLASS HANDLE 2	4402	4409	4416	4423	4430	4437	4444
CLASS HANDLE 3	4403	4410	4417	4424	4431	4438	4445
CLASS HANDLE 4	4404	4411	4418	4425	4432	4439	4446
CLASS HANDLE 5	4405	4412	4419	4426	4433	4440	4447
CLASS HANDLE 6	4406	4413	4420	4427	4434	4441	4448
CLASS HANDLE 7	4407	4414	4421	4428	4435	4442	4449

4400

FIG. 254

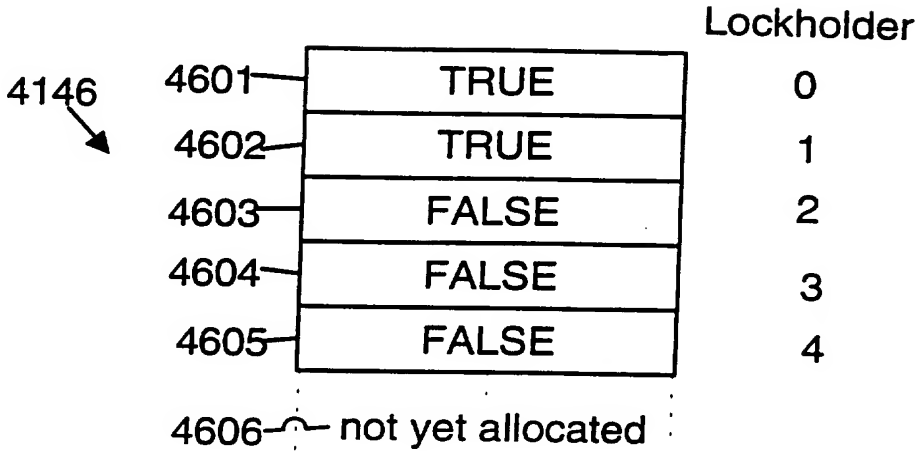


FIG. 255

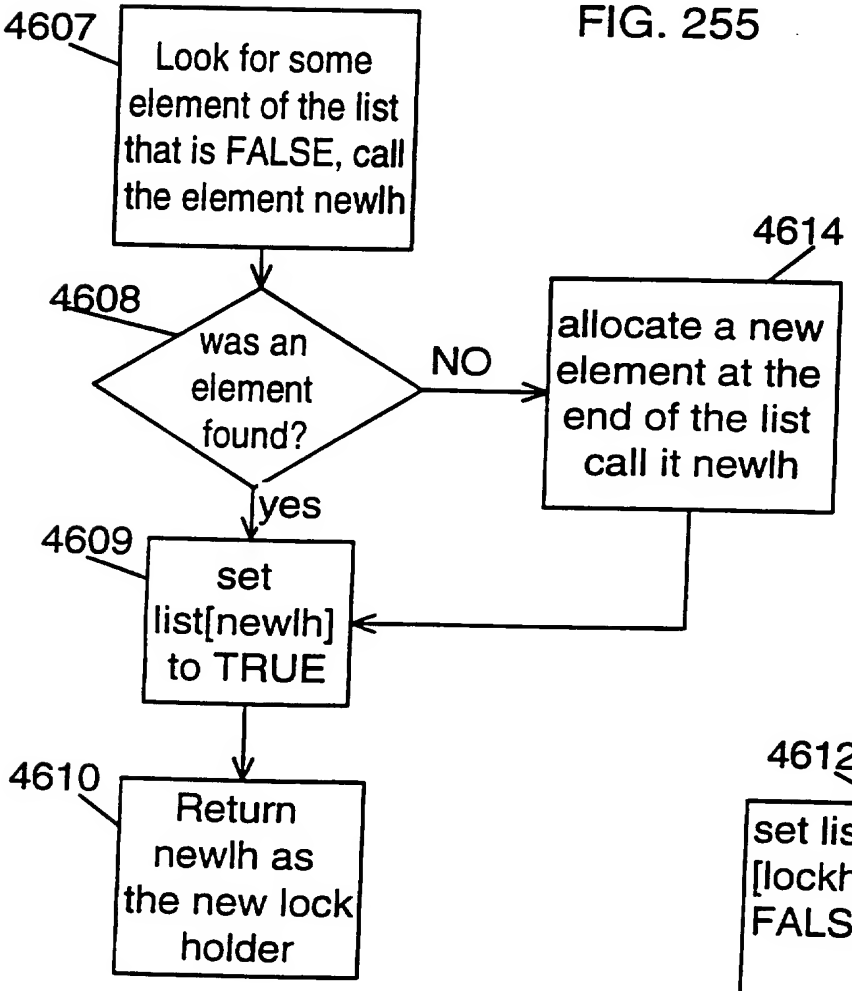


FIG. 256

FIG. 257

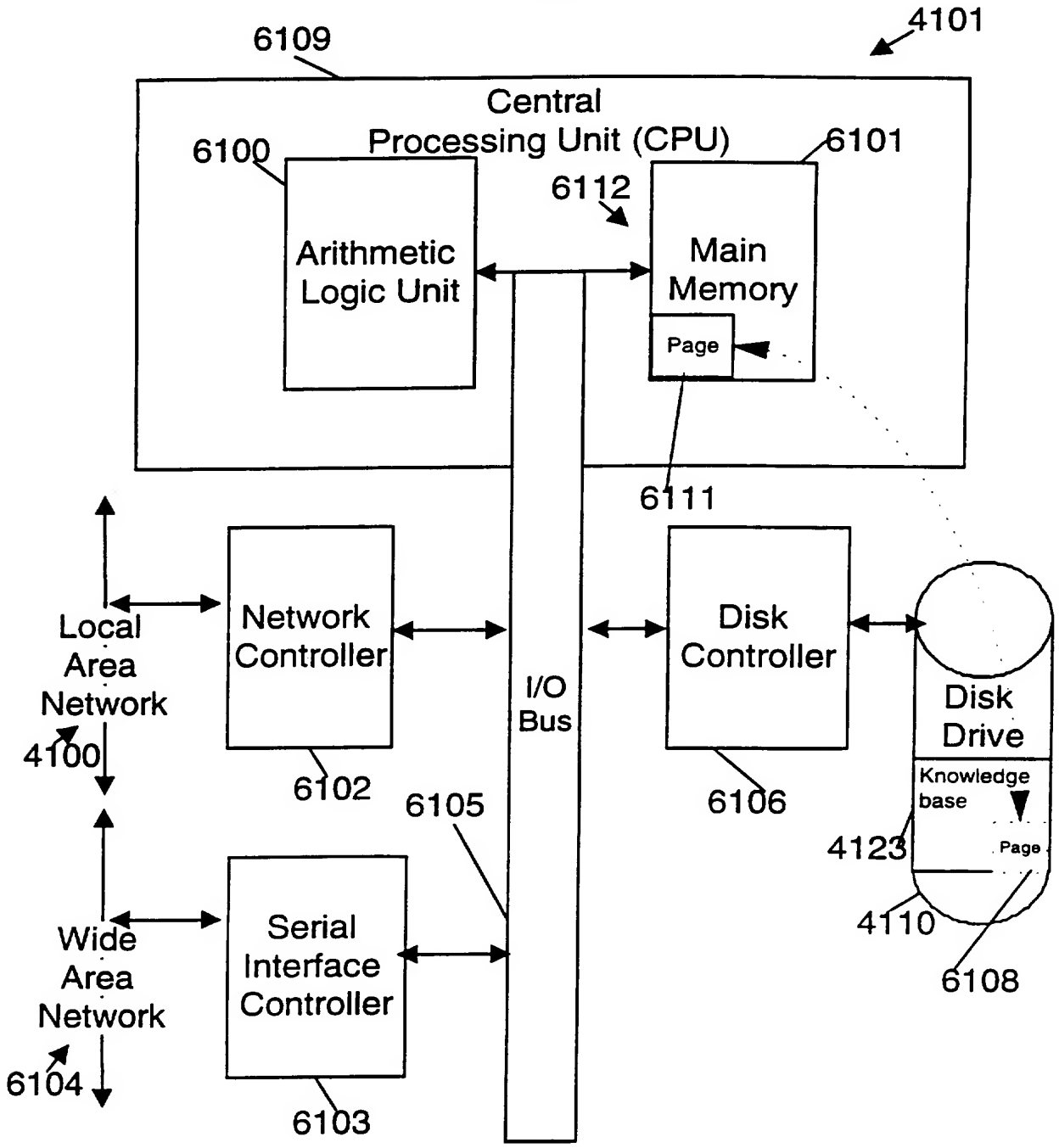

4101
6109
Central
Processing Unit (CPU)
6100
6101
6112
Arithmetic
Logic Unit
Main
Memory
Page
6111
Local
Area
Network
4100
Network
Controller
6102
I/O
Bus
Disk
Controller
6106
Disk
Drive
Knowledge
base
Page
4123
4110
6108
6105
Wide
Area
Network
6104
Serial
Interface
Controller
6103

FIG. 258

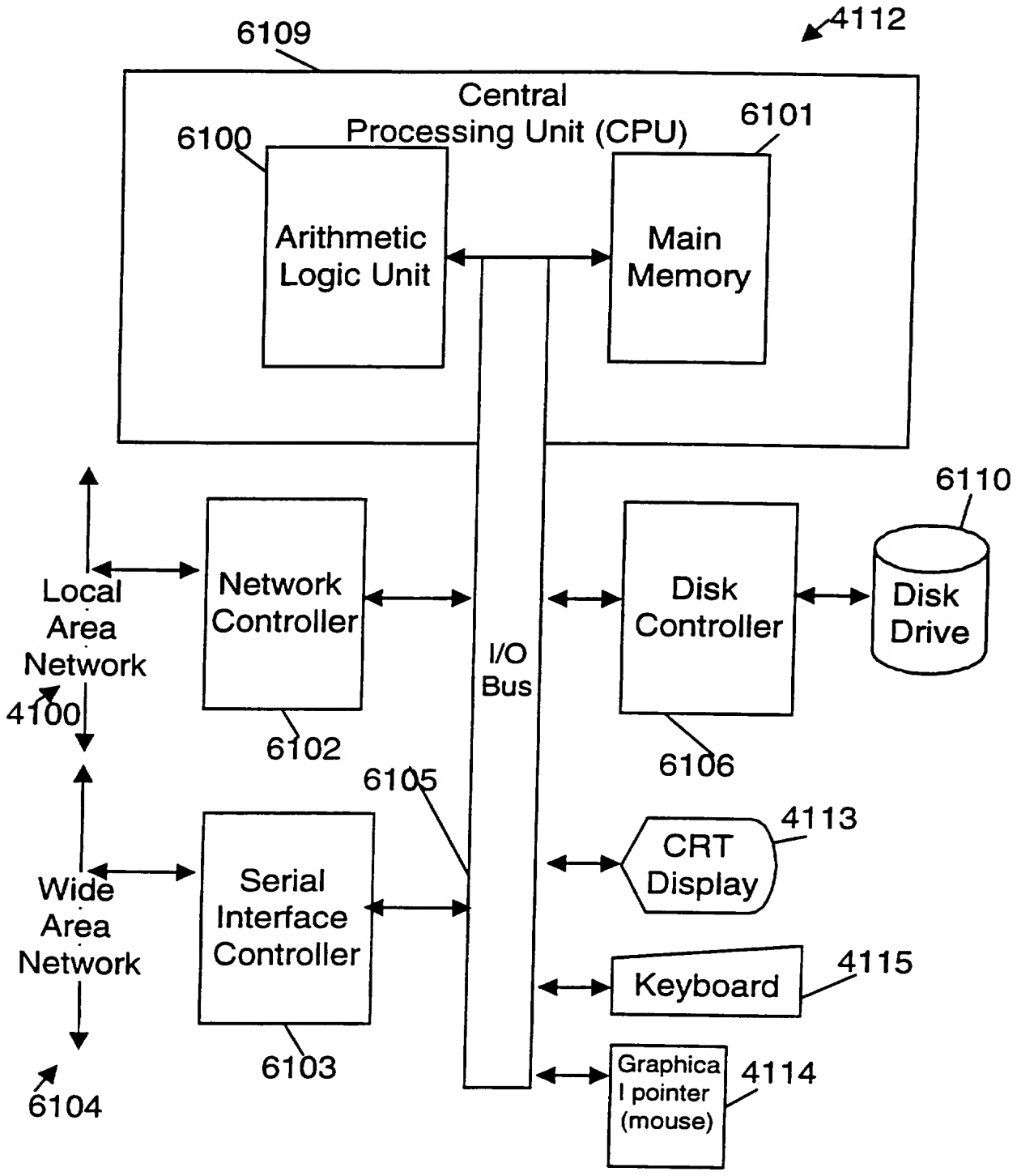

6109
4112
Central
Processing Unit (CPU)
6101
6100
Arithmetic
Logic Unit
Main
Memory
6110
Local
Area
Network
4100
Network
Controller
6102
I/O
Bus
Disk
Controller
6106
Disk
Drive
6105
4113
CRT
Display
Wide
Area
Network
Serial
Interface
Controller
6103
Keyboard
4115
6104
Graphica
l pointer
(mouse)
4114

FIG. 259

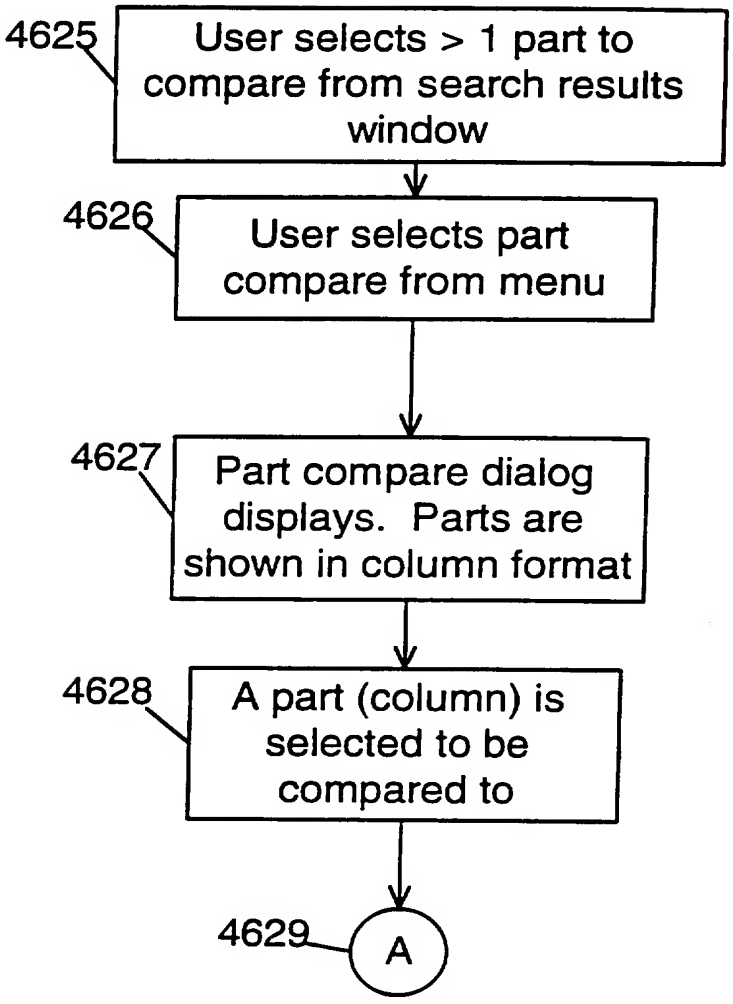

4625
User selects > 1 part to compare from search results window
4626
User selects part compare from menu
4627
Part compare dialog displays. Parts are shown in column format
4628
A part (column) is selected to be compared to
4629
A

FIG. 260

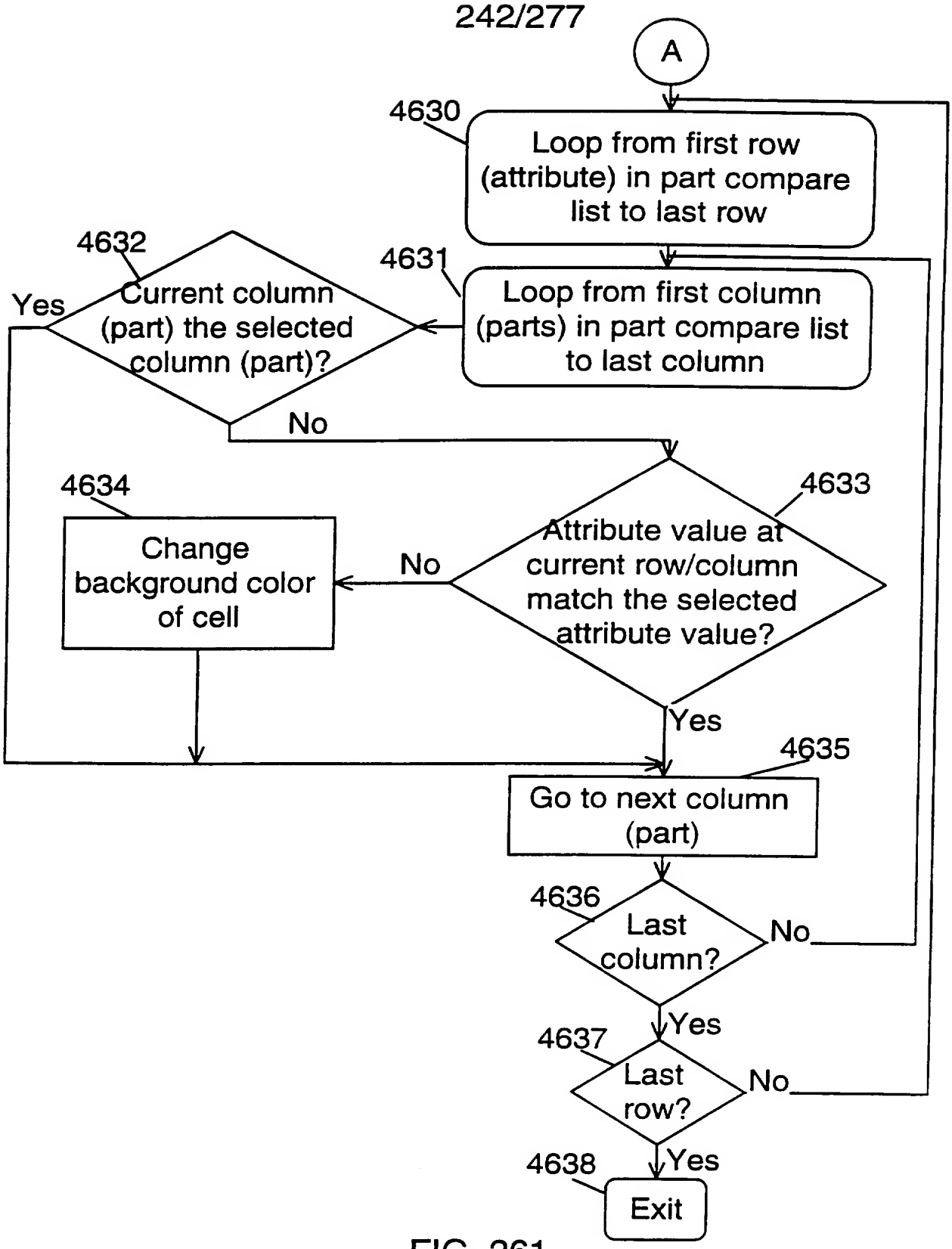

A
4630
Loop from first row (attribute) in part compare list to last row
4631
Loop from first column (parts) in part compare list to last column
4632
Current column (part) the selected column (part)?
Yes
No
4633
Attribute value at current row/column match the selected attribute value?
No
4634
Change background color of cell
Yes
4635
Go to next column (part)
4636
Last column?
No
Yes
4637
Last row?
No
Yes
4638
Exit

FIG. 261

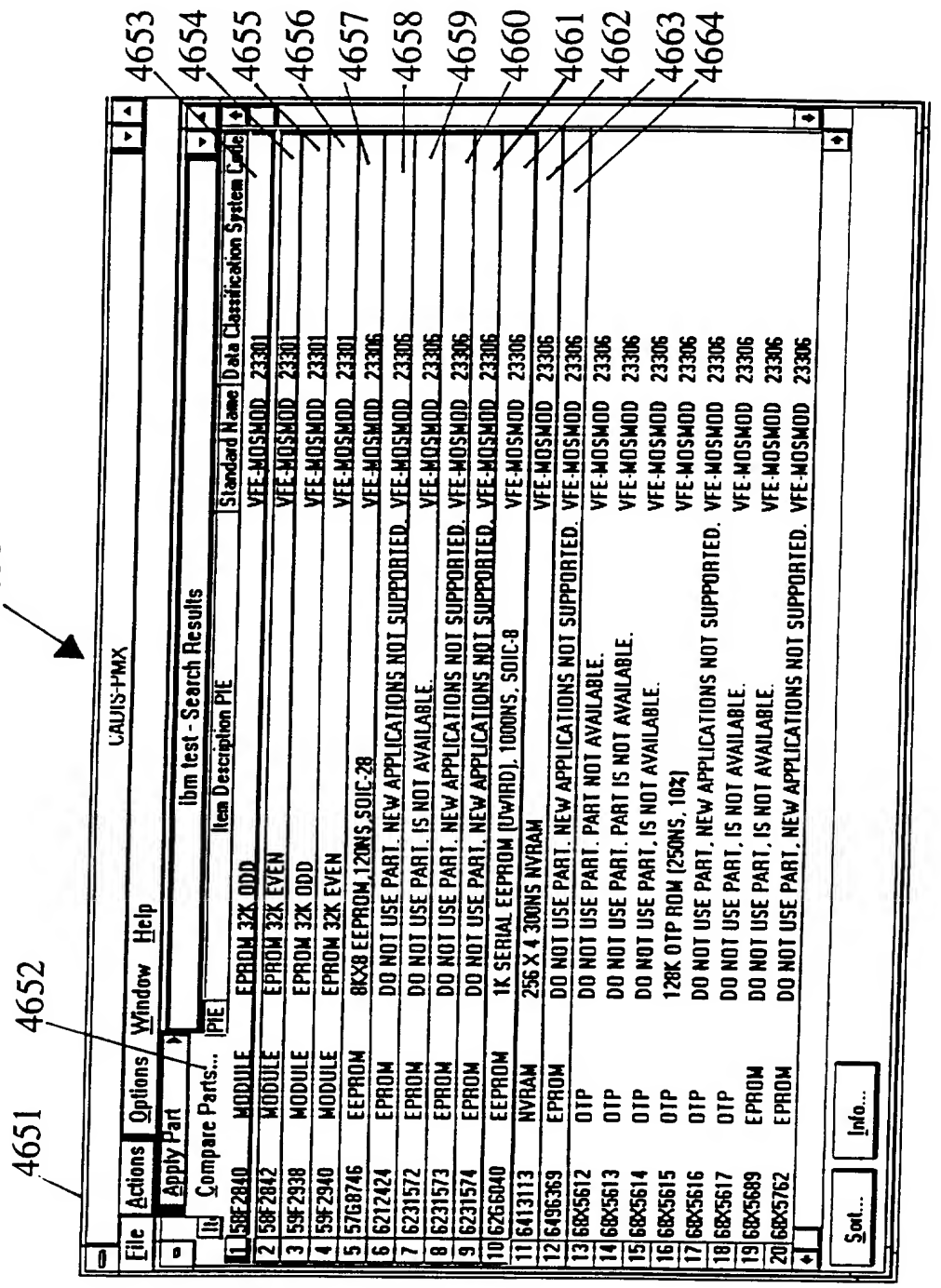

4650
4651
4652
4653
4654
4655
4656
4657
4658
4659
4660
4661
4662
4663
4664
CAUIS-PMX
File Actions Options Window Help
Apply Part
Compare Parts...
Ibm test - Search Results
PIE
Item Description PIE
Standard Name
Data Classification System Code
Sort...
Info...

FIG. 262A

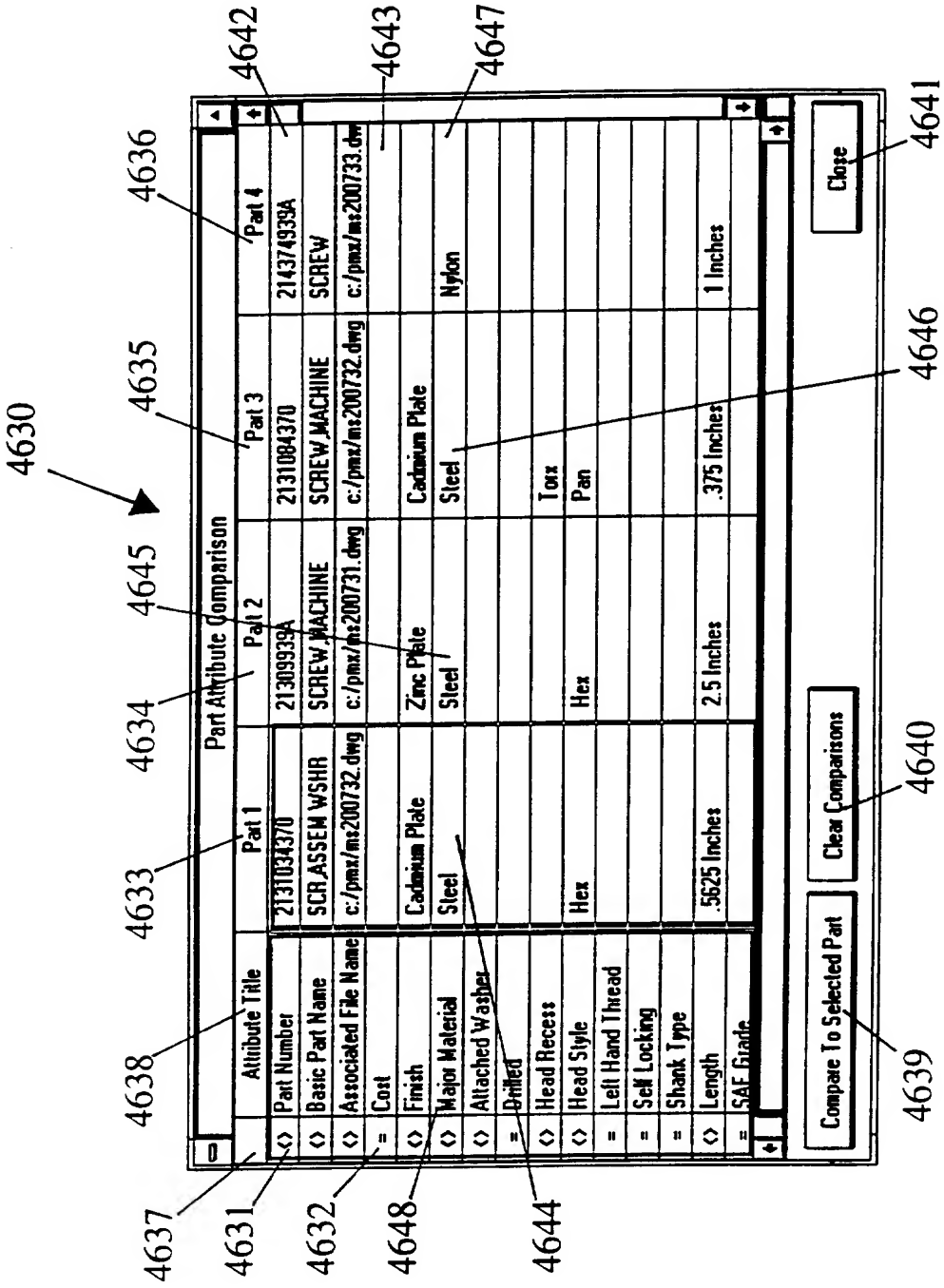

4630
Part Attribute Comparison
Attribute Title
Part 1
Part 2
Part 3
Part 4
Part Number
2131034370
21309939A
2131084370
21437493 9A
Basic Part Name
SCR,ASSEM W/SHR
SCREW,MACHINE
SCREW,MACHINE
SCREW
Associated File Name
c:/pmx/ms200732.dwg
c:/pmx/ms200731.dwg
c:/pmx/ms200732.dwg
c:/pmx/ms200733.dw
Cost
Finish
Cadmium Plate
Zinc Plate
Cadmium Plate
Nylon
Major Material
Steel
Steel
Steel
Attached Washer
Drilled
Head Recess
Torx
Head Style
Hex
Hex
Pan
Left Hand Thread
Self Locking
Shank Type
Length
.5625 Inches
2.5 Inches
.375 Inches
1 Inches
SAE Grade
Compare To Selected Part
Clear Comparisons
Close
4631
4632
4633
4634
4635
4636
4637
4638
4639
4640
4641
4642
4643
4644
4645
4646
4647
4648

FIG. 262B

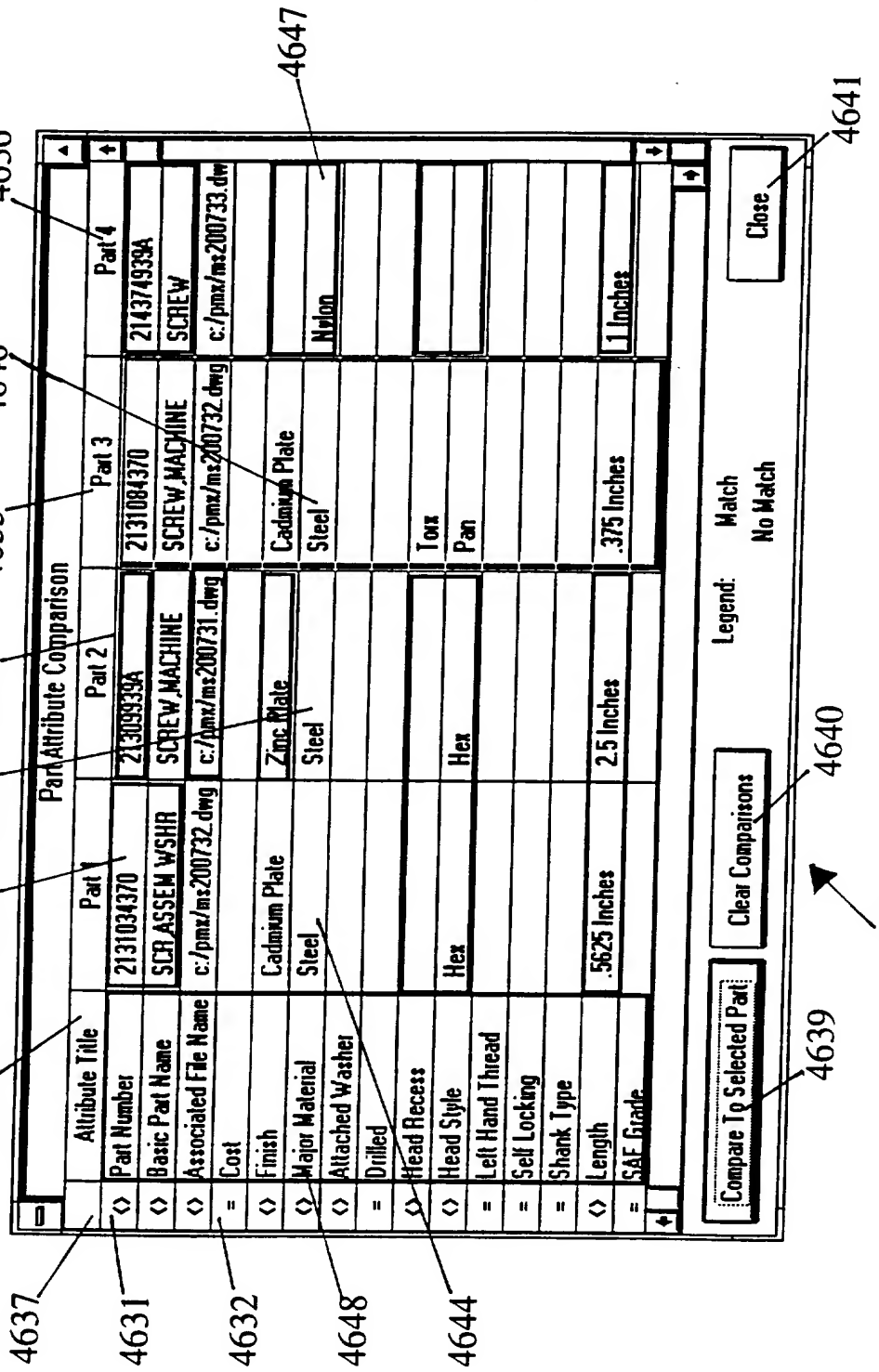

Part Attribute Comparison
Attribute Title
Part 1
Part 2
Part 3
Part 4
Part Number
2131034370
2130939A
2131084370
2143749398
Basic Part Name
SCR ASSEM W/SHR
SCREW MACHINE
SCREW MACHINE
SCREW
Associated File Name
c:/pmx/ms200732.dwg
c:/pmx/ms200731.dwg
c:/pmx/ms200732.dwg
c:/pmx/ms200733.dw
Cost
Finish
Cadmium Plate
Zinc Plate
Cadmium Plate
Nylon
Major Material
Steel
Steel
Steel
Attached Washer
Drilled
Head Recess
Torx
Head Style
Hex
Hex
Pan
Left Hand Thread
Self Locking
Shank Type
Length
.5625 Inches
2.5 Inches
.375 Inches
1 Inches
SAE Grade
Compare To Selected Part
Clear Comparisons
Legend:
Match
No Match
Close
4630
4631
4632
4633
4634
4635
4636
4637
4638
4639
4640
4641
4644
4645
4646
4647
4648

FIG. 263

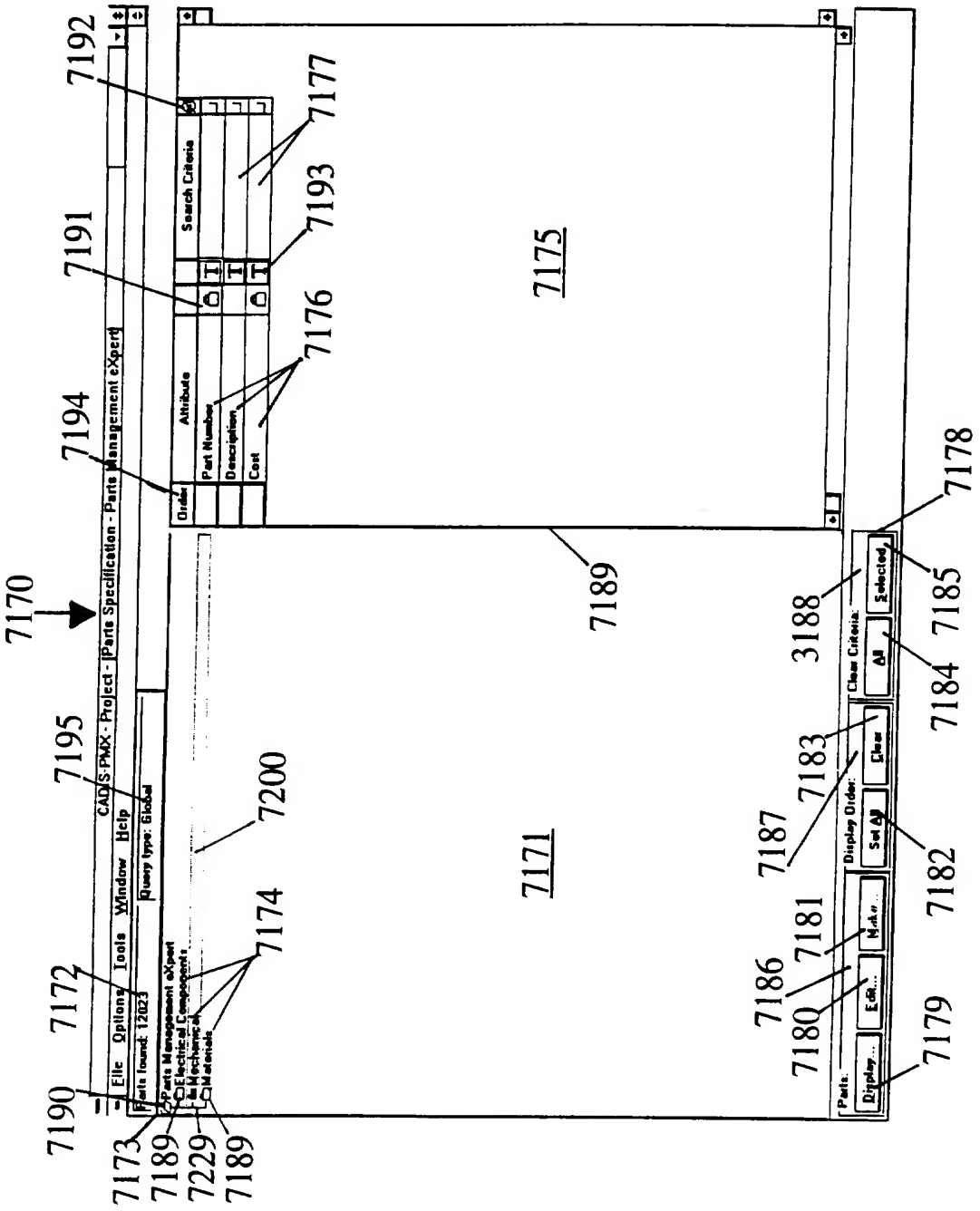

7192
7177
7193
7191
7175
7176
7194
7178
7189
7170
3188
7185
7195
7184
7200
7183
7187
7171
7174
7182
7181
7172
7186
7180
7179
7190
7173
7189
7229
7189
CADS-PMX - Project - [Parts Specification - Parts Management eXpert]
File Options Tools Window Help
Search Criteria
Attribute
Part Number
Description
Cost
Order
Parts found: 12023
Query type: Global
Parts Management eXpert
Electrical Component
Mechanical
Materials
Clear Criteria:
All
Selected
Display Order:
Clear
Set All
Edit...
Marker...
Parts:
Display...

FIG. 264

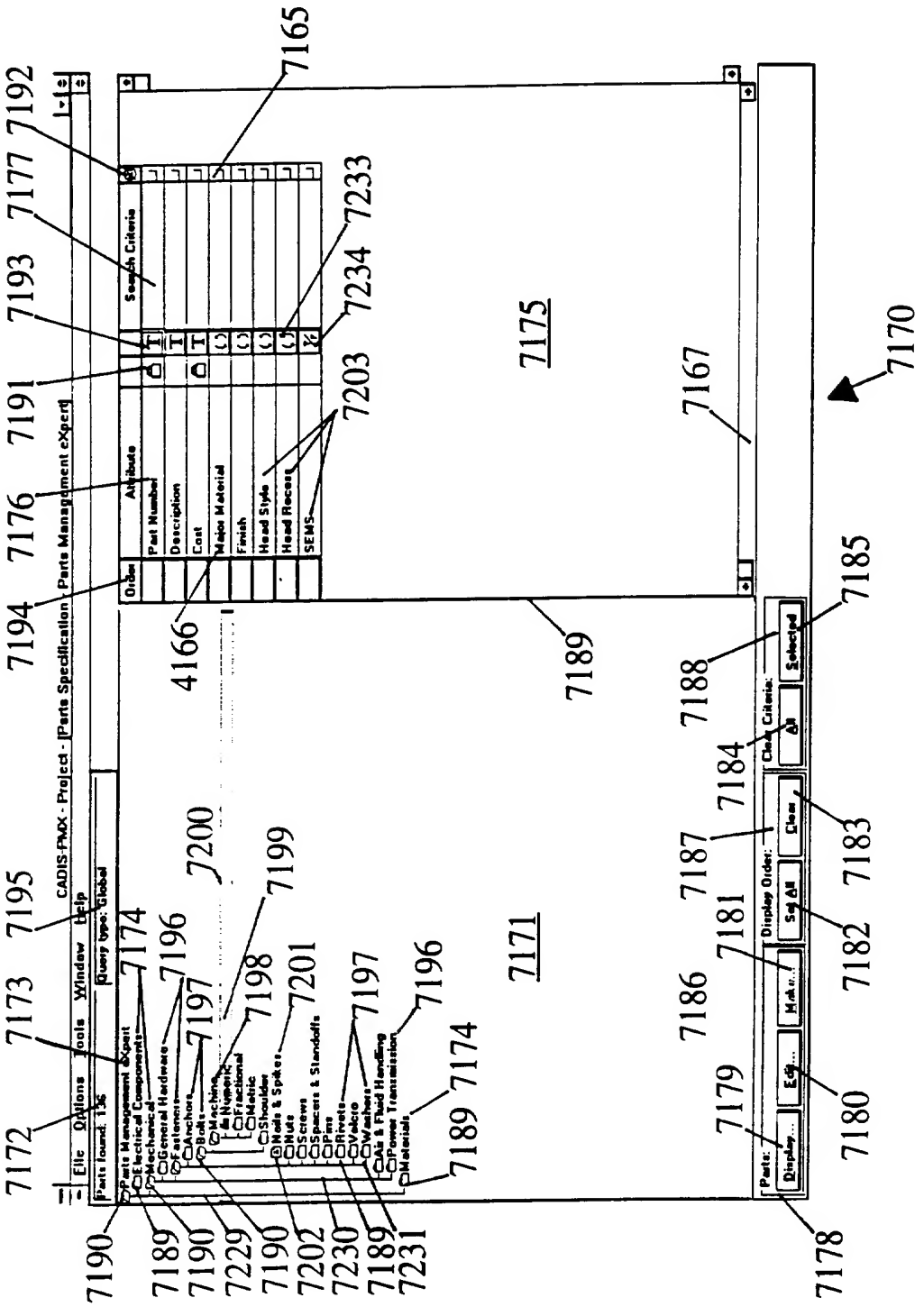

7165
7192
7177
7193
7191
7176
7194
7233
7234
7203
7175
7167
7170
4166
7189
7188
7185
7184
7187
7183
7181
7182
7186
7179
7180
7178
7171
7174
7196
7199
7198
7201
7197
7202
7230
7231
7200
7195
7173
7172
7190
7229
Parts Management eXpert
Mechanical Components
Electrical Components
General Hardware
Fasteners
Anchors
Nuts
Washers
Hexnuts
Nylocks
Keps
Metric
Shoulder
Nuts & Spikes
Screws
Spacers & Standoffs
Pins
Rivets
Velcro
Washers
Air & Fluid Handling
Power Transmission
Materials
CADIS-FMAX - Project - [Parts Specification - Parts Management eXpert]
File Options Tools Window Help
Parts found: 136
Query type: Global
Order
Attribute
Part Number
Description
Cost
Major Material
Finish
Head Style
Head Recess
SEMS
Search Criteria
Class Criteria:
All
Selected
Display Order:
Clear
Set All
Parts:
Display...
Edit...
Make...

FIG. 265

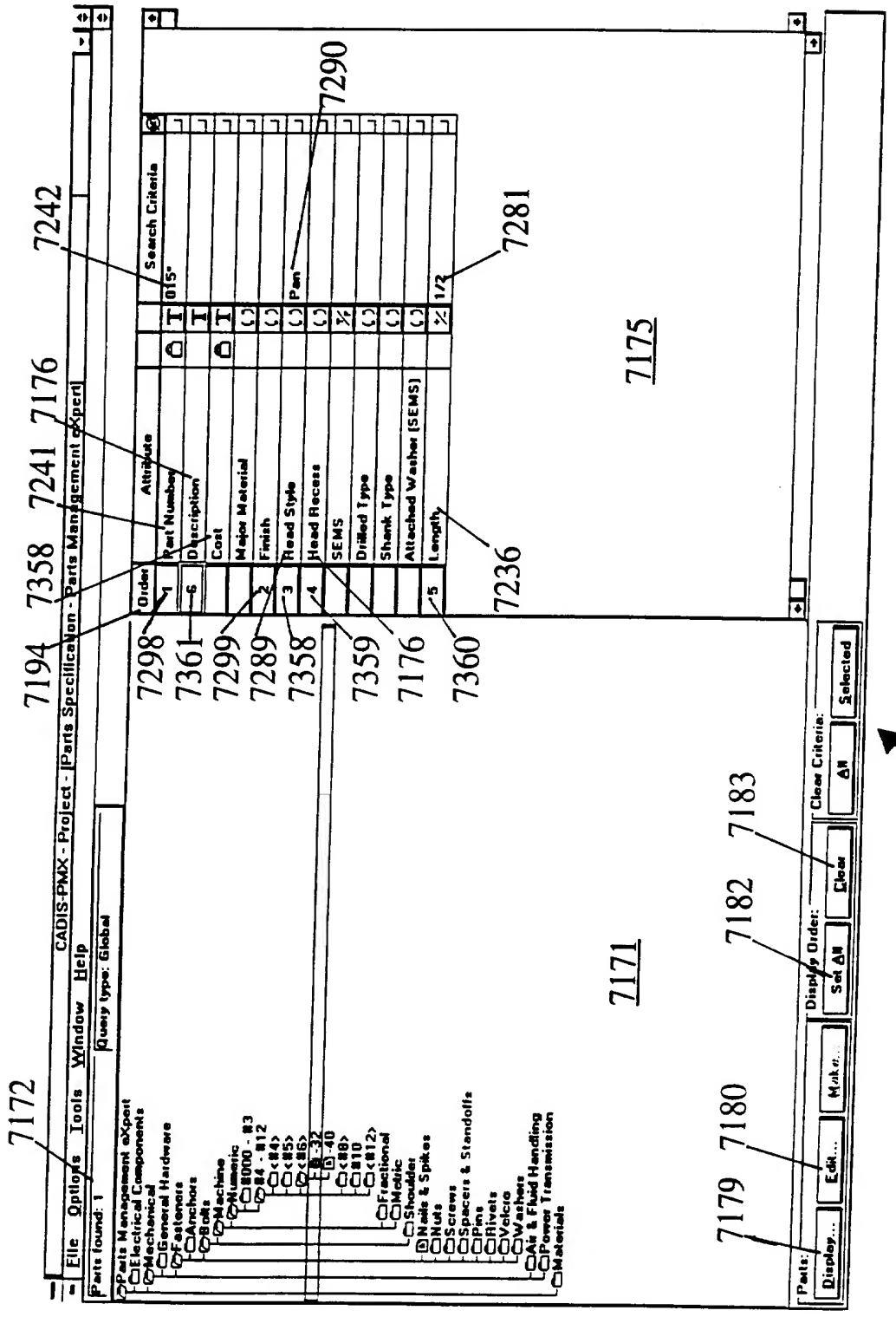

7172
7194
7358
7241
7176
7242
7290
7281
7175
7236
7298
7361
7299
7289
7358
7359
7176
7360
7171
7182
7183
7180
7179
7170
CADIS-PMX - Project - [Parts Specification - Parts Management eXpert]
File Options Tools Window Help
Parts found: 1
Query type: Global
Parts Management eXpert
Electrical Components
Mechanical
General Hardware
Fasteners
Anchors
Bolts
Machine
Numeric
#000 - #3
#4 - #12
<#4>
<#5>
<#6>
#-32
<#8>
#10
<#12>
Fractional
Metric
Shoulder
Nails & Spikes
Nuts
Screws
Spacers & Standoffs
Pins
Rivets
Velcro
Washers
Air & Fluid Handling
Power Transmission
Materials
Order
Attribute
Search Criteria
Part Number
015*
Description
Cost
Major Material
Finish
Head Style
Pan
Head Recess
SEMS
Drilled Type
Shank Type
Attached Washer (SEMS)
Length
1/2
Display Order:
Set All
Clear
Clear Criteria:
All
Selected
Parts:
Display...
Edit...
Make...

FIG. 266

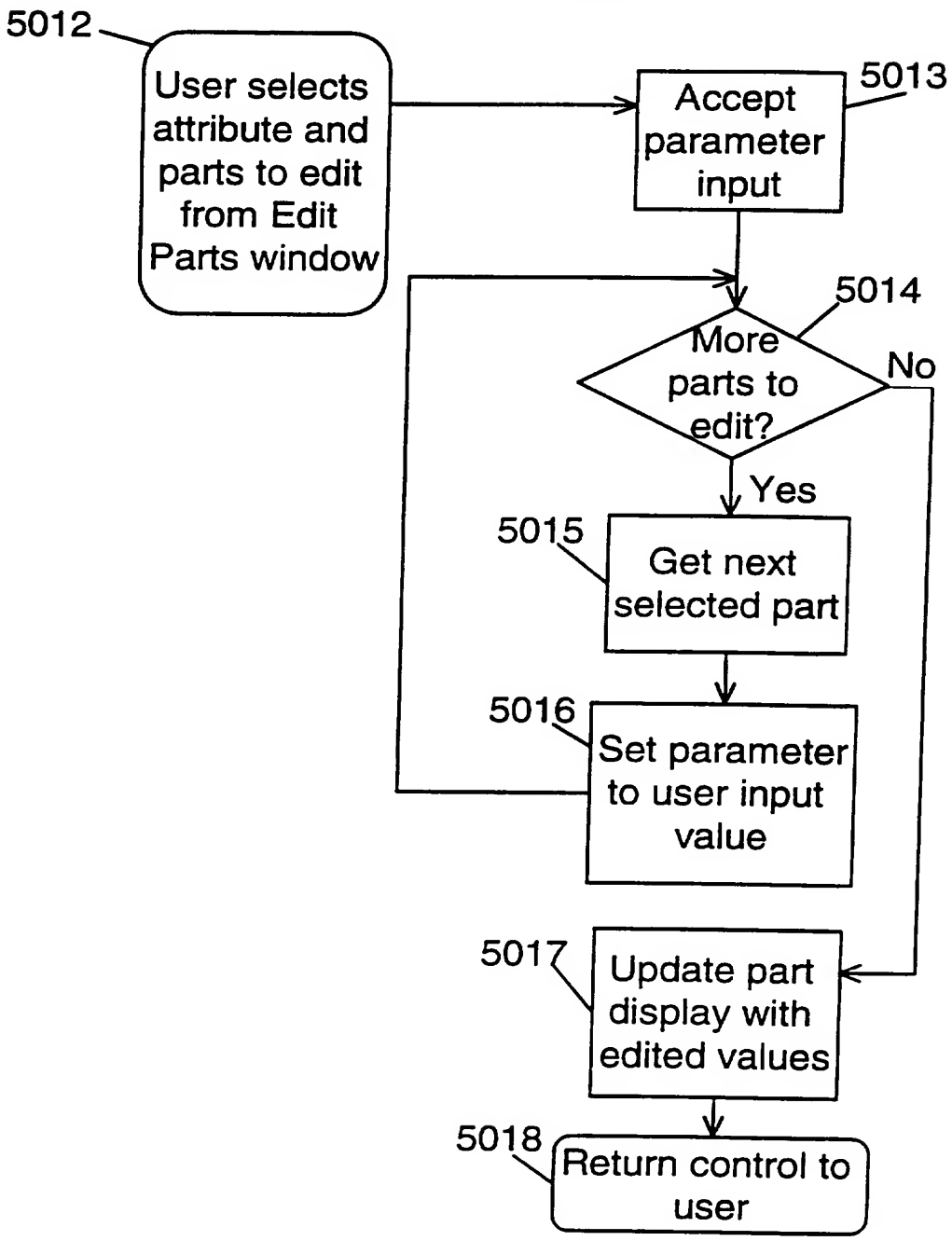

5012
User selects attribute and parts to edit from Edit Parts window
Accept parameter input
5013
5014
More parts to edit?
No
Yes
5015
Get next selected part
5016
Set parameter to user input value
5017
Update part display with edited values
5018
Return control to user

FIG. 267

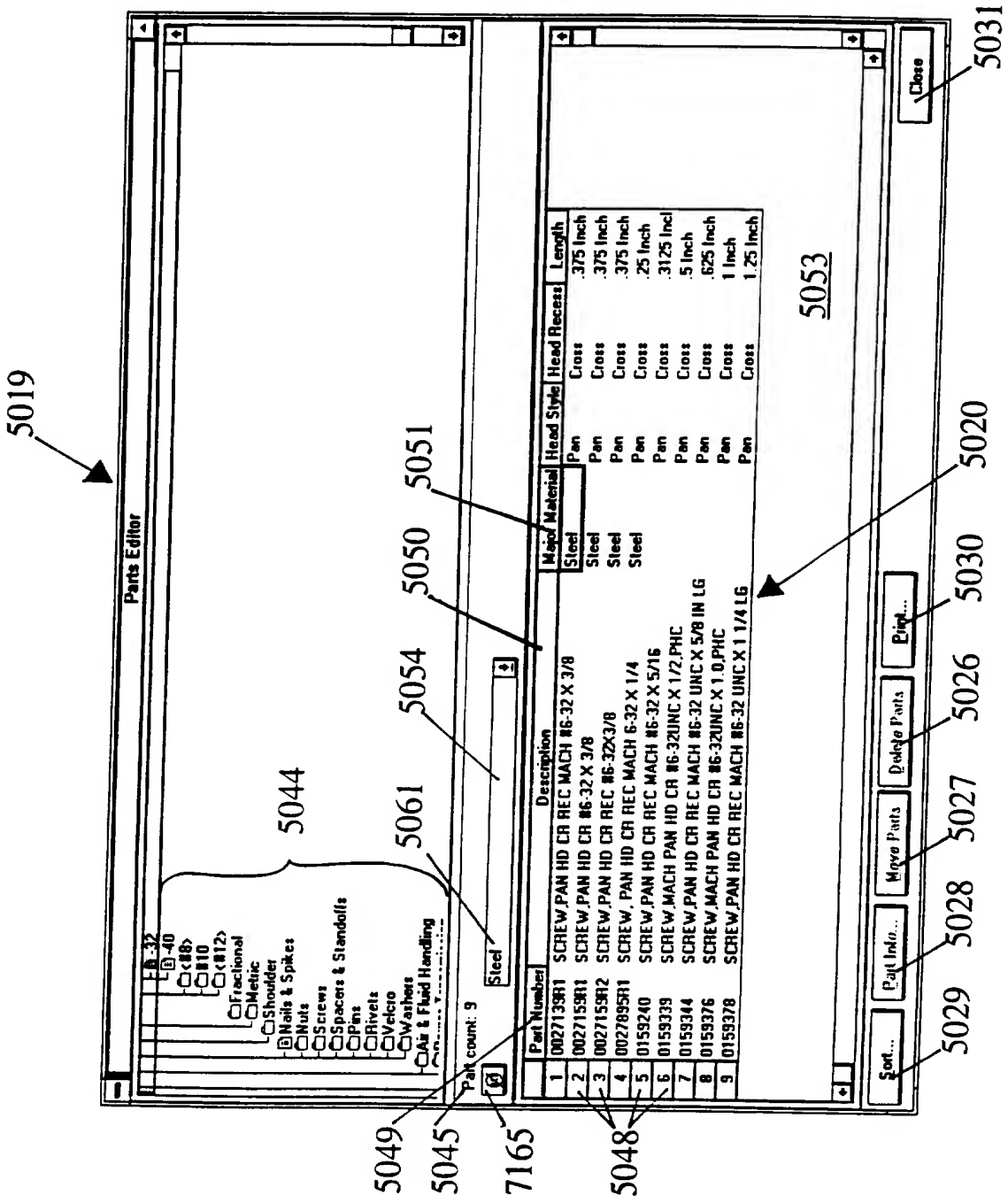

5019
5020
5026
5027
5028
5029
5030
5031
5044
5045
5048
5049
5050
5051
5053
5054
5061
7165
Parts Editor
Part count: 9
Steel
Close
Sort...
Part Info...
Move Parts
Delete Parts
Print...

FIG. 268

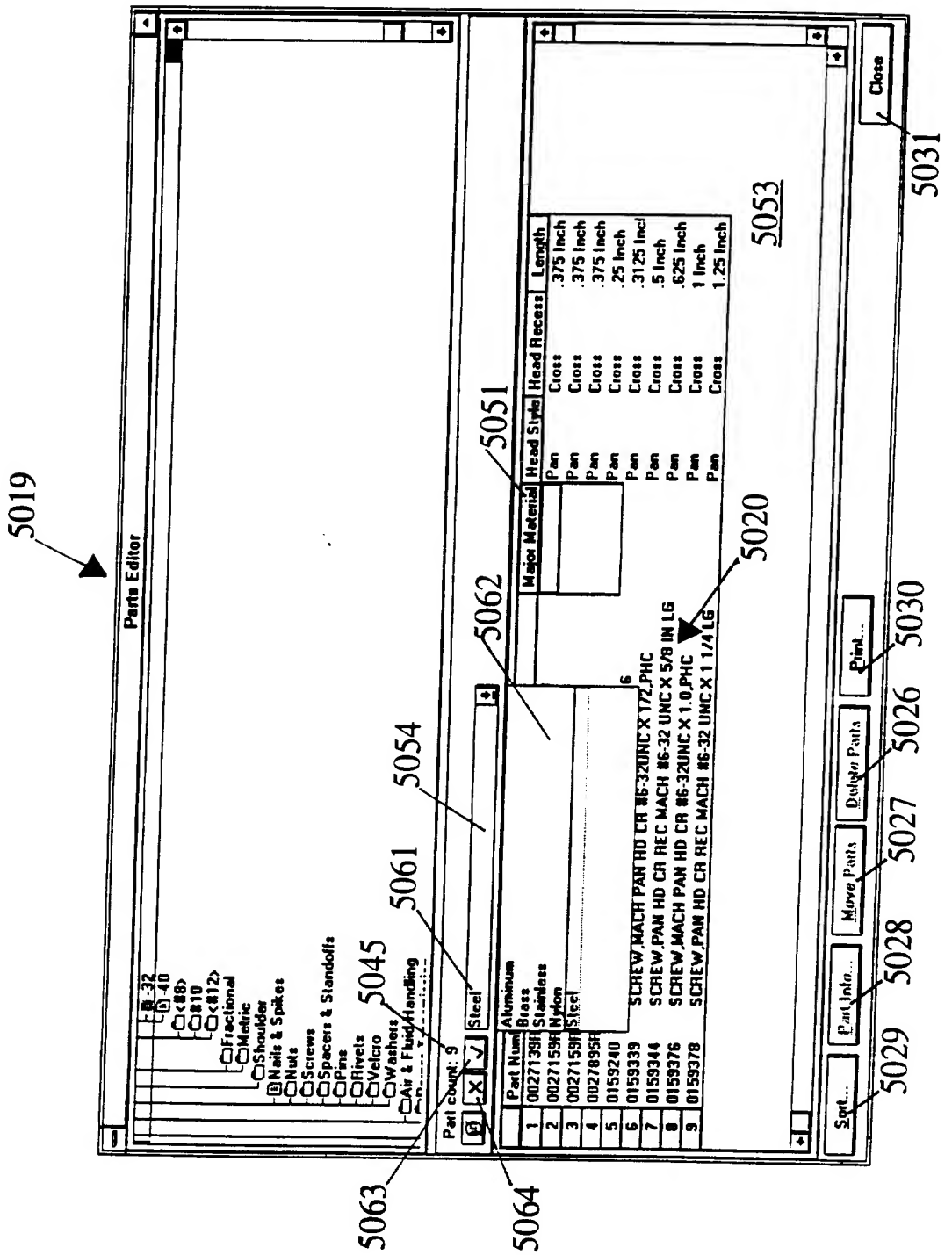

5019
Parts Editor
Fractional
Metric
Shoulder
Nuts
Screws
Pins
Rivets
Velcro
Washers
Steel
Part count: 9
Aluminum
Brass
Stainless
Nylon
Steel
Part Num
Major Material
Head Style
Head Recess
Length
Pan
Cross
.375 Inch
.25 Inch
.3125 Inch
.5 Inch
.625 Inch
1 Inch
1.25 Inch
SCREW,MACH PAN HD CR #6-32UNC X 1/2,PHC
SCREW,PAN HD CR REC MACH #6-32 UNC X 5/8 IN LG
SCREW,MACH PAN HD CR #6-32UNC X 1.0,PHC
SCREW,PAN HD CR REC MACH #6-32 UNC X 1 1/4 LG
Close
Print...
Delete Parts
Move Parts
Part Info...
Sort...
5020
5030
5026
5027
5028
5029
5031
5051
5053
5054
5061
5062
5063
5064
5045

FIG. 269

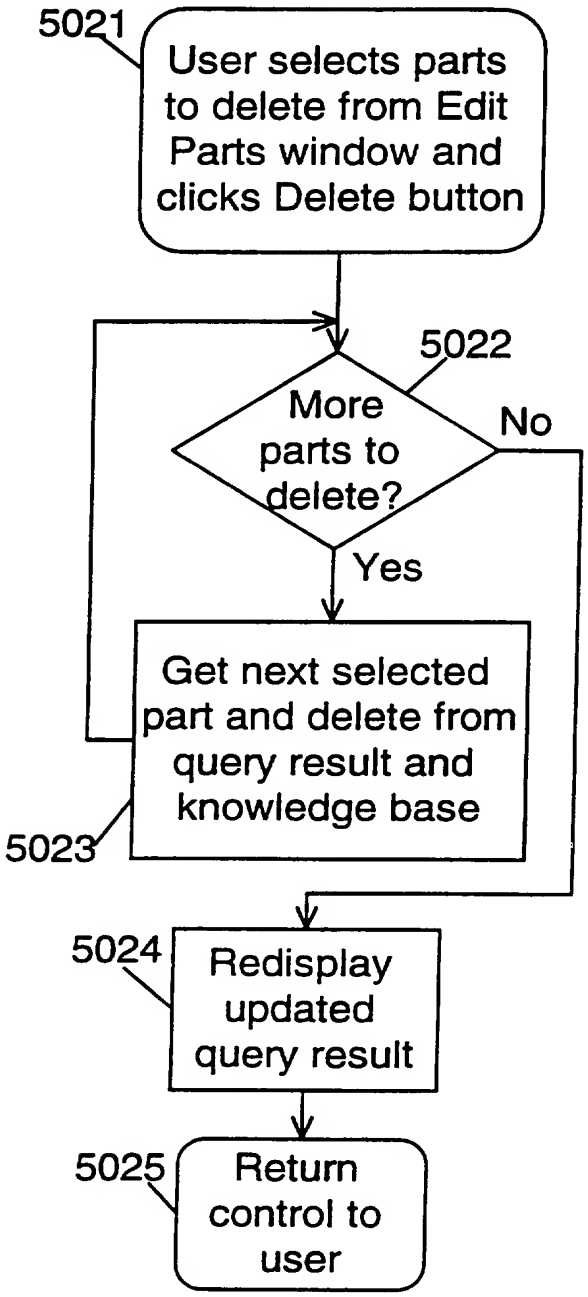

5021
User selects parts to delete from Edit Parts window and clicks Delete button
5022
More parts to delete?
No
Yes
Get next selected part and delete from query result and knowledge base
5023
5024
Redisplay updated query result
5025
Return control to user

FIG. 270

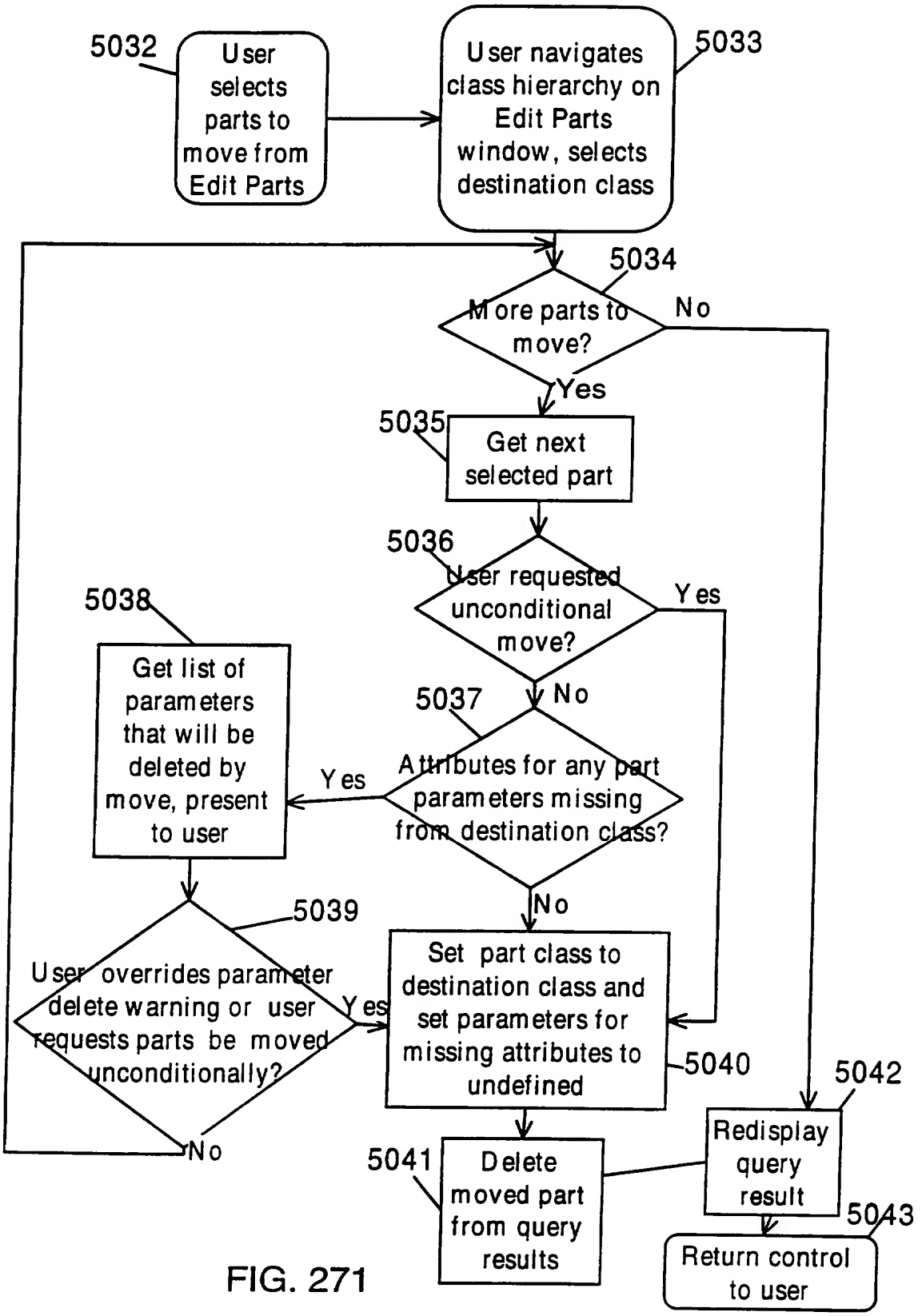

5032
User selects parts to move from Edit Parts
User navigates class hierarchy on Edit Parts window, selects destination class
5033
5034
More parts to move?
No
Yes
5035
Get next selected part
5036
User requested unconditional move?
Yes
5038
Get list of parameters that will be deleted by move, present to user
5037
No
Attributes for any part parameters missing from destination class?
Yes
No
5039
User overrides parameter delete warning or user requests parts be moved unconditionally?
Yes
No
Set part class to destination class and set parameters for missing attributes to undefined
5040
5042
Redisplay query result
5041
Delete moved part from query results
5043
Return control to user

FIG. 271

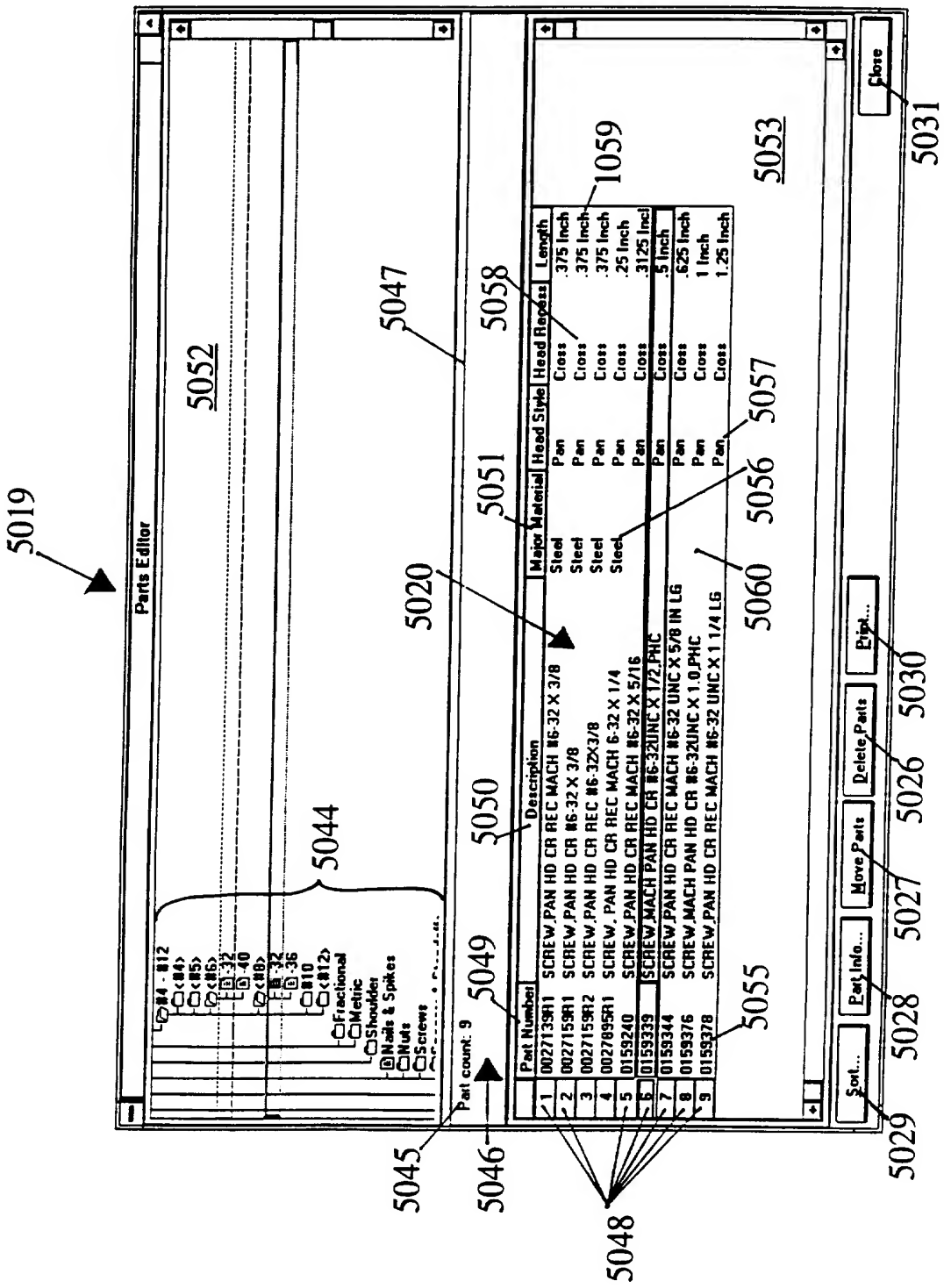

5019
Parts Editor
5052
5047
5058
5051
5020
5050
5044
5049
5045
5046
5048
5053
1059
5057
5056
5060
5030
5026
5027
5028
5029
5055
5031
Close
Print...
Delete Parts
Move Parts
Part Info...
Sort...
Part count: 9
Description
Part Number
Major Material
Head Style
Head Recess
Length
SCREW,PAN HD CR REC MACH #6-32 X 3/8
SCREW,PAN HD CR #6-32 X 3/8
SCREW,PAN HD CR REC #6-32X3/8
SCREW, PAN HD CR REC MACH 6-32 X 1/4
SCREW,PAN HD CR REC MACH #6-32 X 5/16
SCREW,MACH PAN HD CR #6-32UNC X 1/2 PHC
SCREW,PAN HD CR REC MACH #6-32 UNC X 5/8 IN LG
SCREW,MACH PAN HD CR #6-32UNC X 1.0 PHC
SCREW,PAN HD CR REC MACH #6-32 UNC X 1 1/4 LG
Steel
Pan
Cross
.375 Inch
.25 Inch
.3125 Inch
.5 Inch
.625 Inch
1 Inch
1.25 Inch

FIG. 272

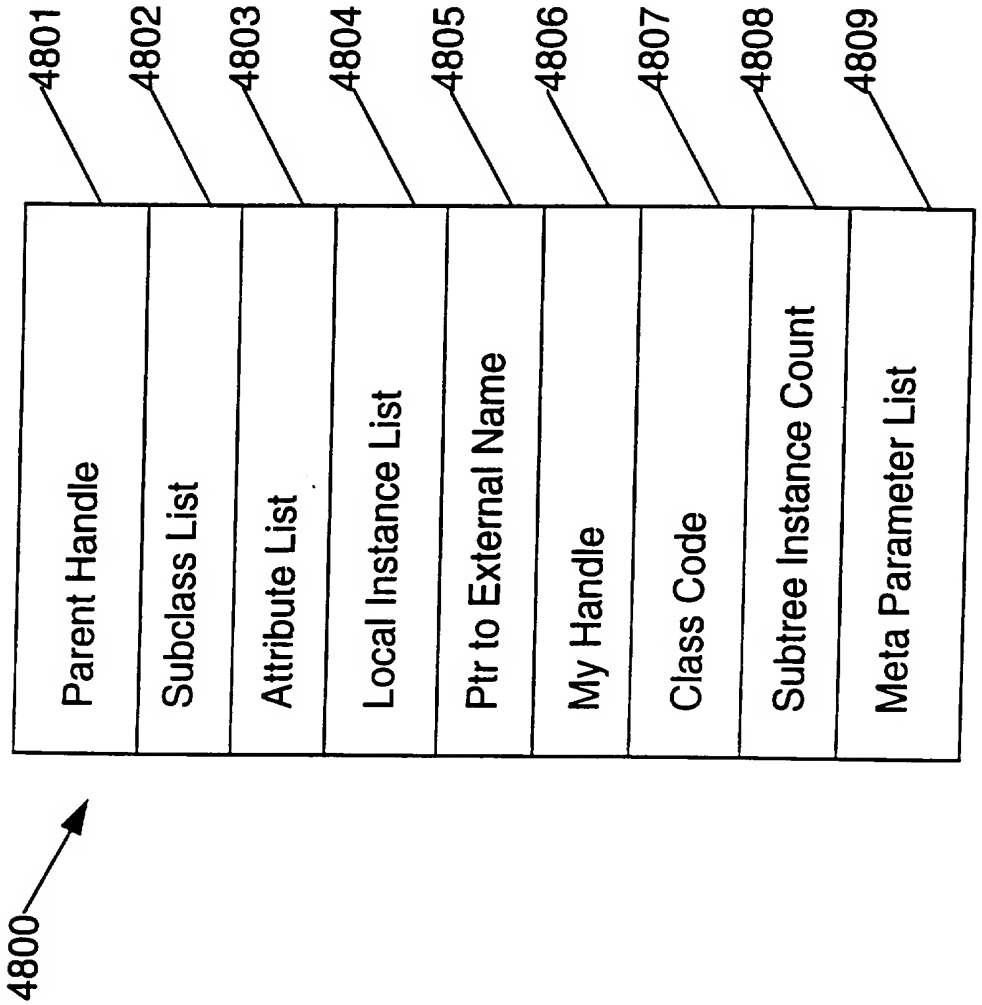

4801
4802
4803
4804
4805
4806
4807
4808
4809
Parent Handle
Subclass List
Attribute List
Local Instance List
Ptr to External Name
My Handle
Class Code
Subtree Instance Count
Meta Parameter List
4800

FIG. 273

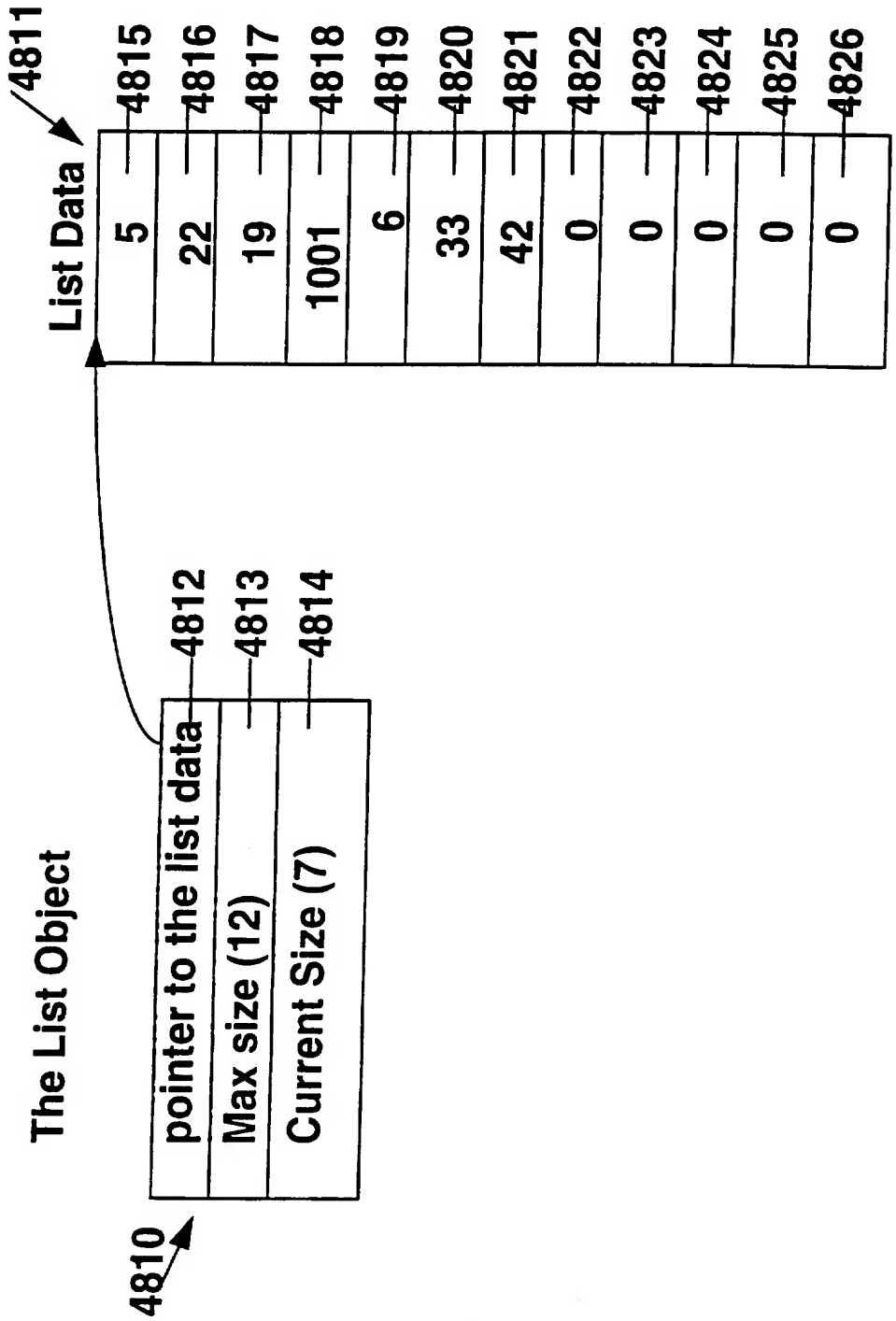

4811
List Data
4815
4816
4817
4818
4819
4820
4821
4822
4823
4824
4825
4826
5
22
19
1001
6
33
42
0
0
0
0
0
The List Object
4812
4813
4814
pointer to the list data
Max size (12)
Current Size (7)
4810

FIG. 274

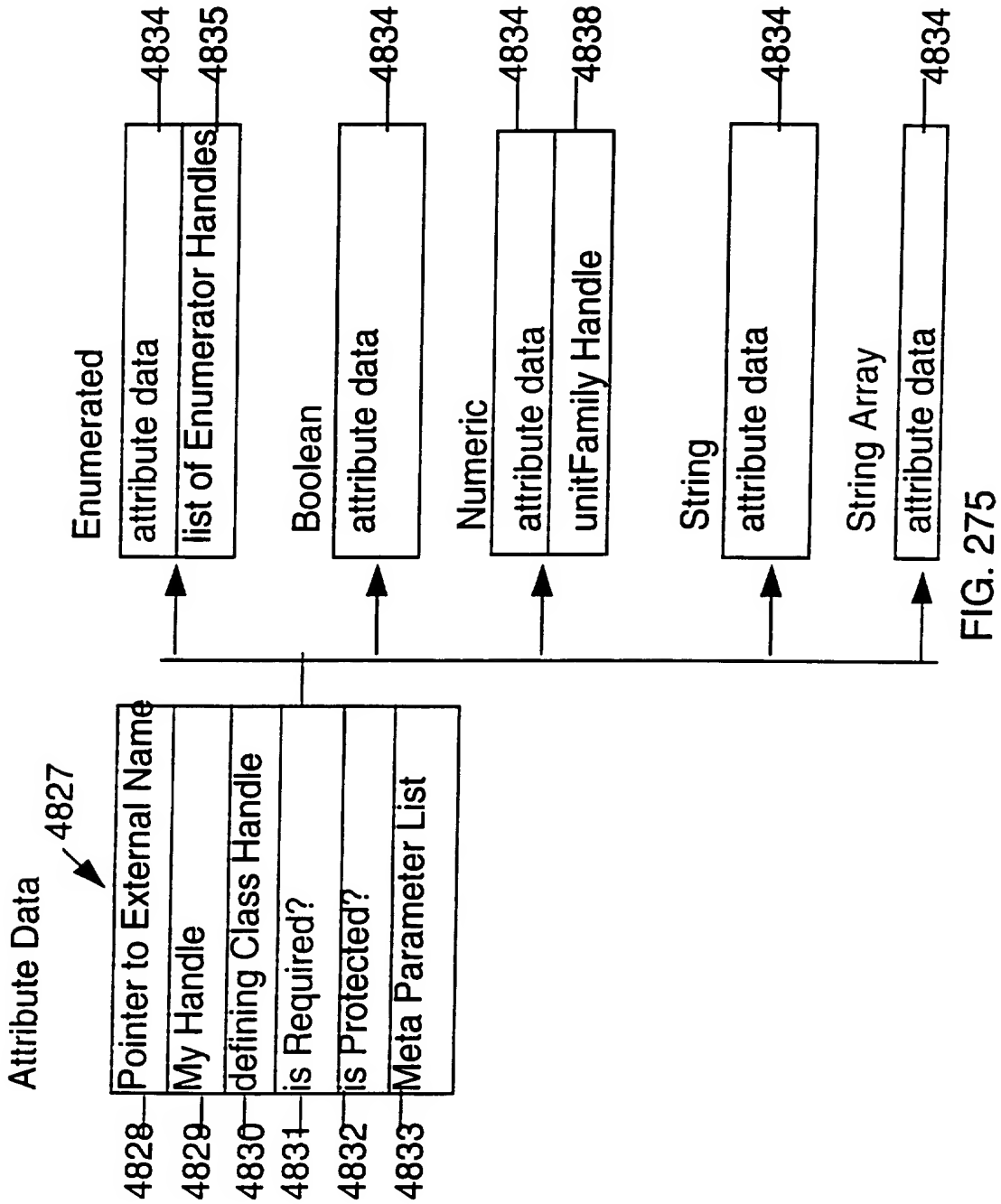

Attribute Data
4827
Pointer to External Name
My Handle
defining Class Handle
is Required?
is Protected?
Meta Parameter List
4828
4829
4830
4831
4832
4833
Enumerated
attribute data
list of Enumerator Handles
4834
4835
Boolean
attribute data
4834
Numeric
attribute data
unitFamily Handle
4834
4838
String
attribute data
4834
String Array
attribute data
4834

FIG. 275

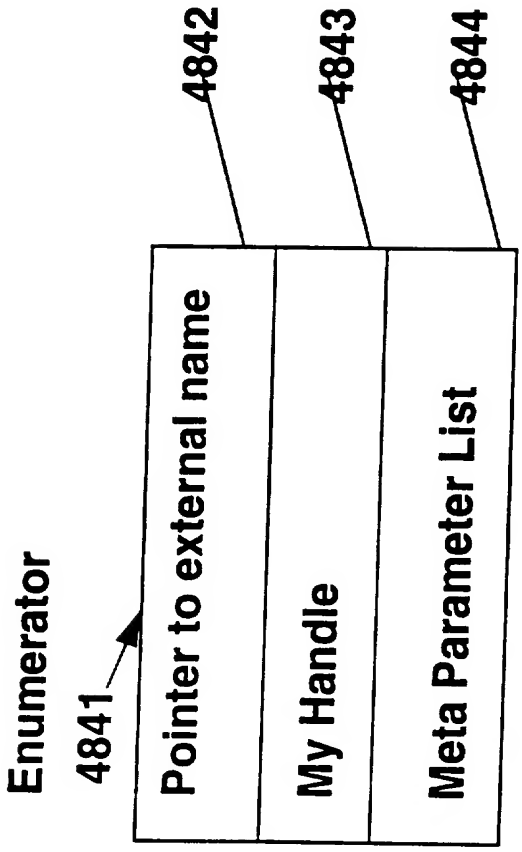

Enumerator
4841
Pointer to external name
My Handle
Meta Parameter List
4842
4843
4844

FIG. 276

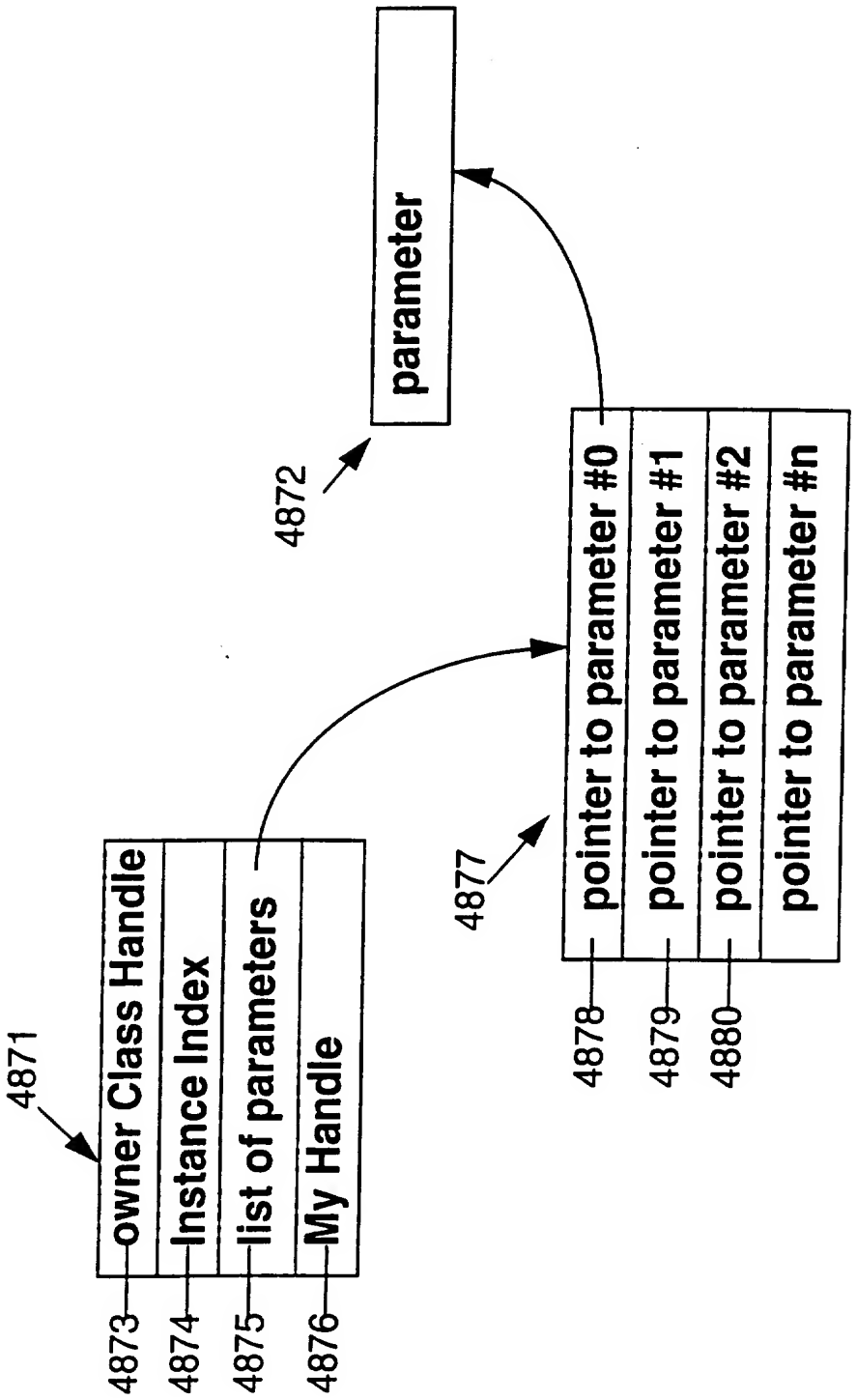

parameter
4872
pointer to parameter #0
pointer to parameter #1
pointer to parameter #2
pointer to parameter #n
4877
4878
4879
4880
4871
owner Class Handle
Instance Index
list of parameters
My Handle
4873
4874
4875
4876

FIG. 277

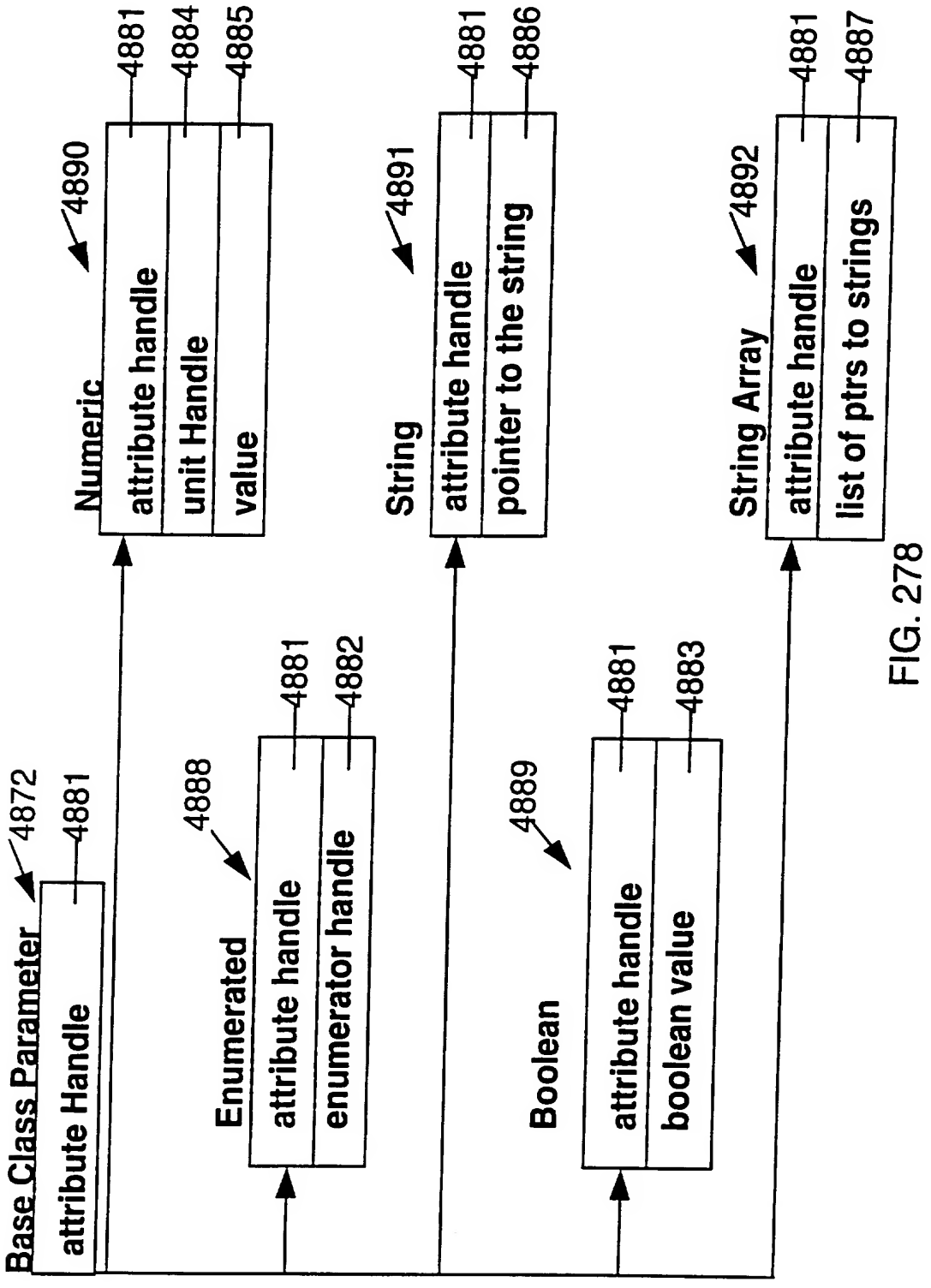

Base Class Parameter
4872
attribute Handle
4881
Numeric
4890
attribute handle
unit Handle
value
4881
4884
4885
Enumerated
4888
attribute handle
enumerator handle
4881
4882
String
4891
attribute handle
pointer to the string
4881
4886
Boolean
4889
attribute handle
boolean value
4881
4883
String Array
4892
attribute handle
list of ptrs to strings
4881
4887

FIG. 278

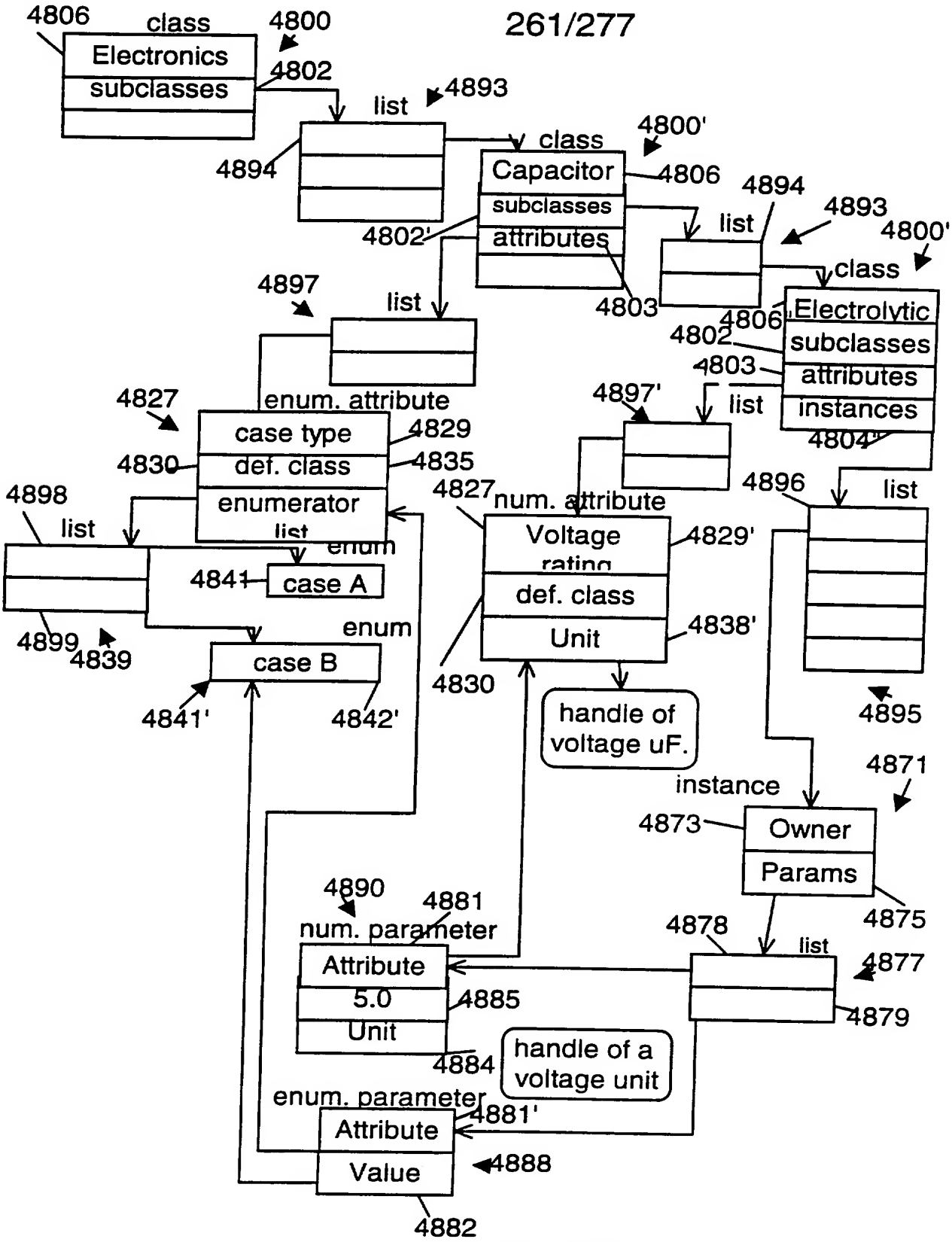

4806
class
Electronics
subclasses
4800
4802
4893
list
4894
4800'
class
Capacitor
subclasses
attributes
4806
4802'
4803
4894
4893
list
4800'
class
Electrolytic
subclasses
attributes
instances
4806
4802
4803
4804'
4897
list
4827
enum. attribute
case type
def. class
enumerator list
4829
4830
4835
4898
list
enum
4841
case A
4899
4839
enum
case B
4841'
4842'
4897'
list
4827
num. attribute
Voltage rating
def. class
Unit
4829'
4838'
4830
handle of voltage uF.
4896
list
4895
4871
instance
4873
Owner
Params
4875
4890
4881
num. parameter
Attribute
5.0
Unit
4885
4884
4878
list
4877
4879
handle of a voltage unit
enum. parameter
4881'
Attribute
Value
4888
4882

FIG. 279

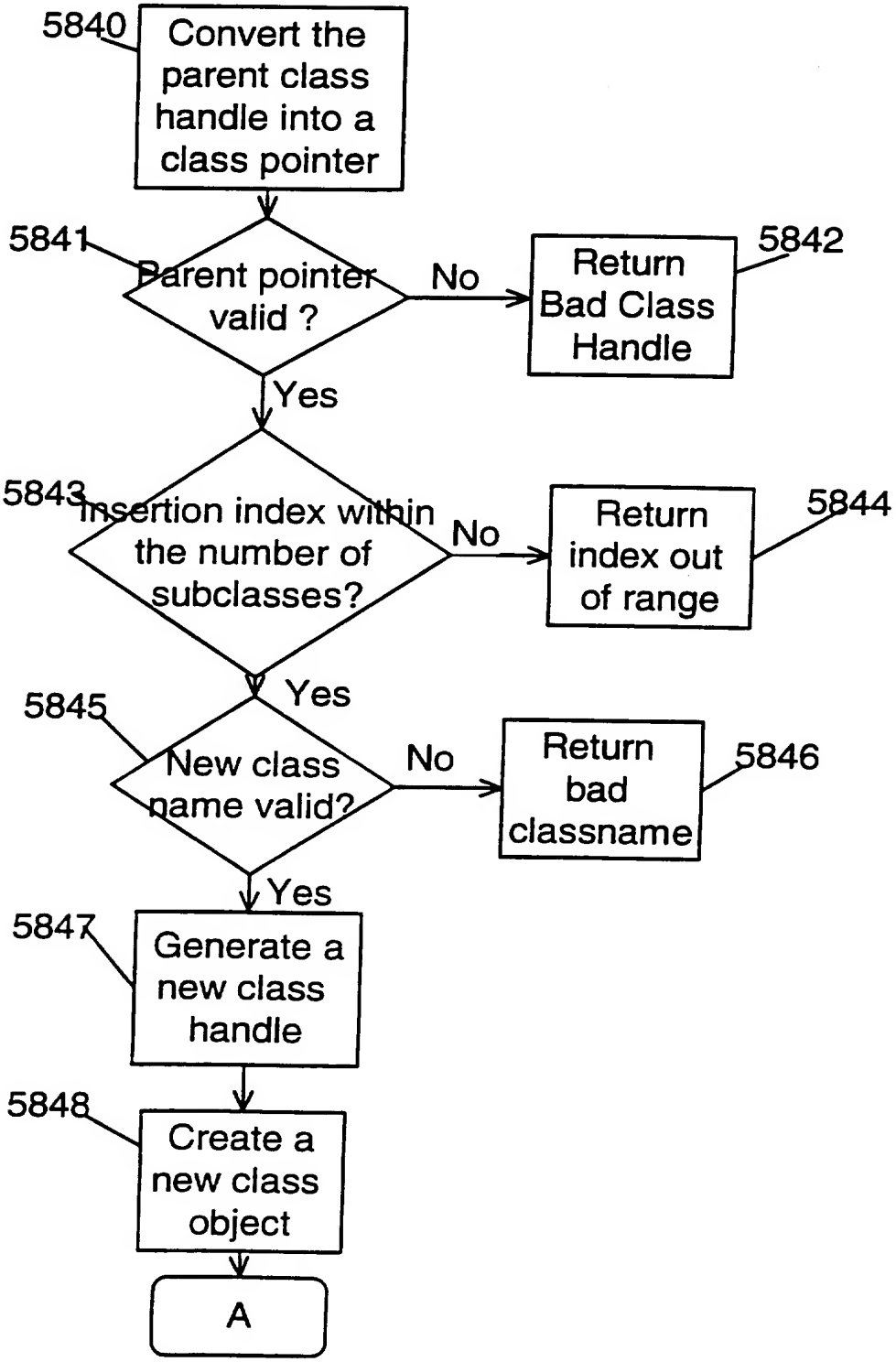

5840
Convert the parent class handle into a class pointer
5841
Parent pointer valid ?
No
5842
Return Bad Class Handle
Yes
5843
Insertion index within the number of subclasses?
No
5844
Return index out of range
Yes
5845
New class name valid?
No
5846
Return bad classname
Yes
5847
Generate a new class handle
5848
Create a new class object
A

FIG. 280

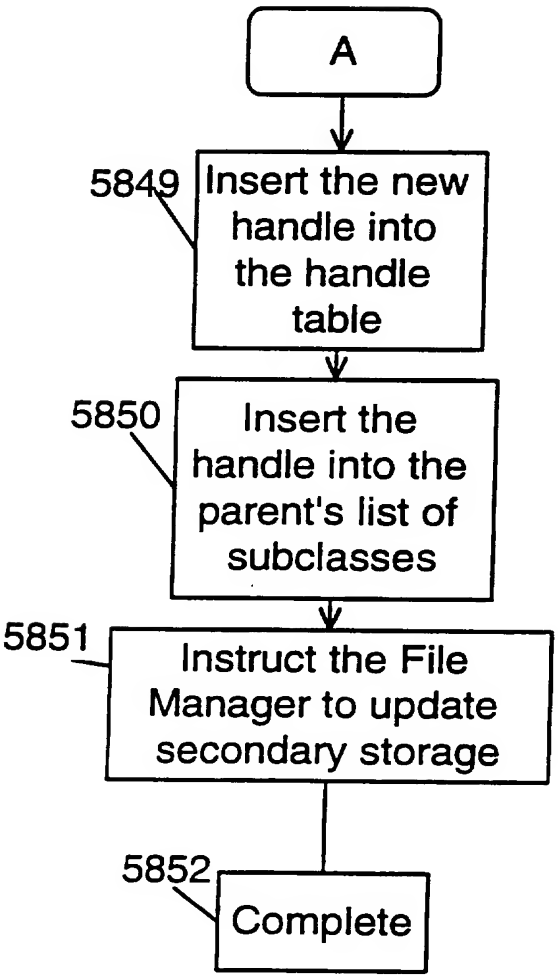

A
5849
Insert the new handle into the handle table
5850
Insert the handle into the parent's list of subclasses
5851
Instruct the File Manager to update secondary storage
5852
Complete

FIG. 281

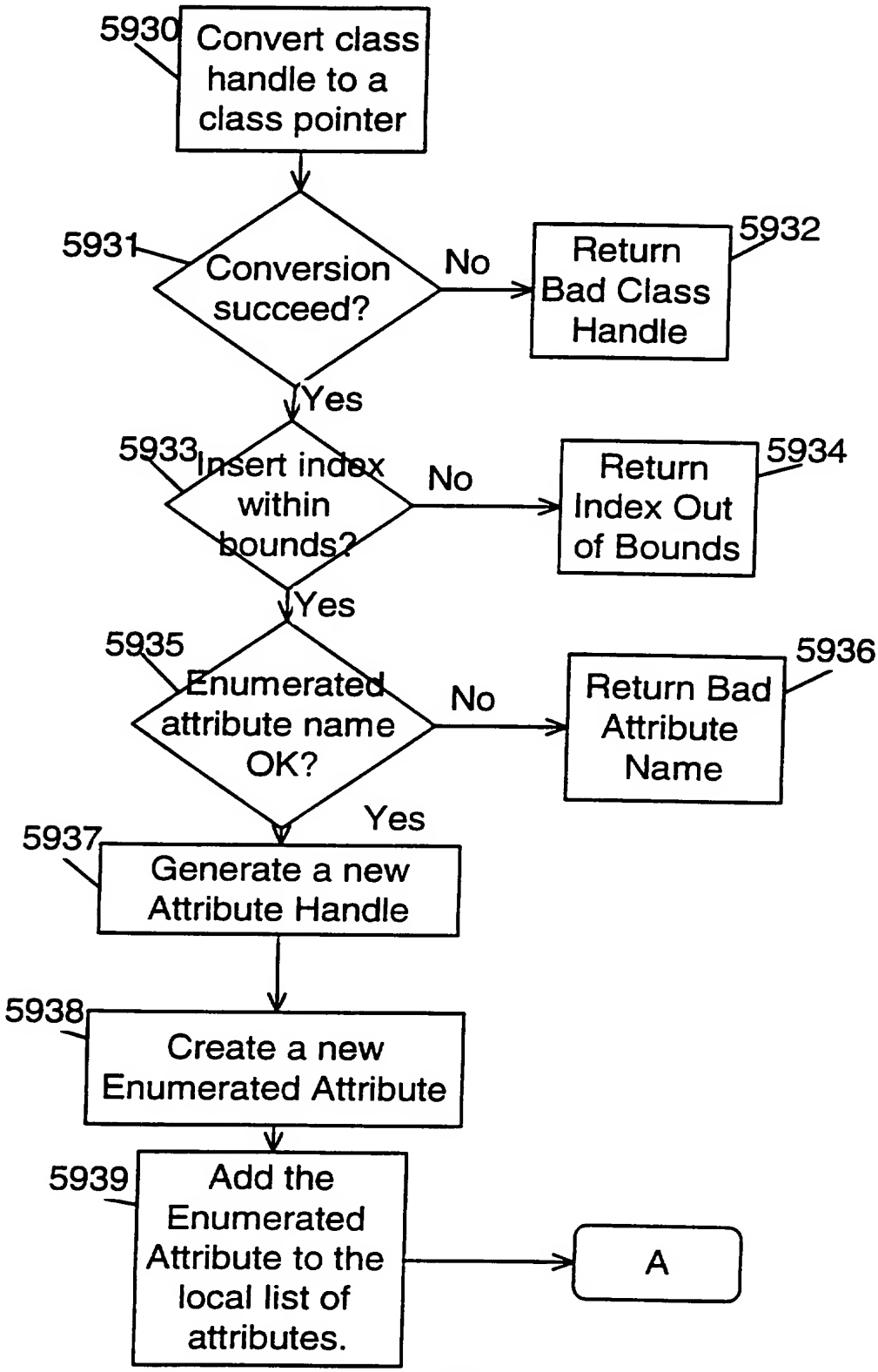

5930
Convert class handle to a class pointer
5931
Conversion succeed?
No
5932
Return Bad Class Handle
Yes
5933
Insert index within bounds?
No
5934
Return Index Out of Bounds
Yes
5935
Enumerated attribute name OK?
No
5936
Return Bad Attribute Name
Yes
5937
Generate a new Attribute Handle
5938
Create a new Enumerated Attribute
5939
Add the Enumerated Attribute to the local list of attributes.
A

FIG. 282

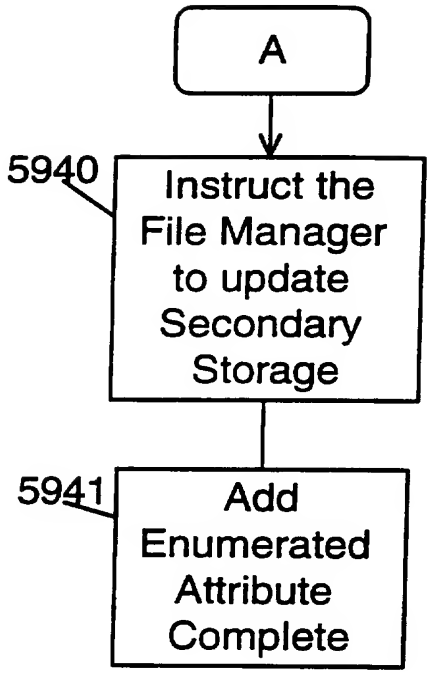

A
5940
Instruct the File Manager to update Secondary Storage
5941
Add Enumerated Attribute Complete

FIG. 283

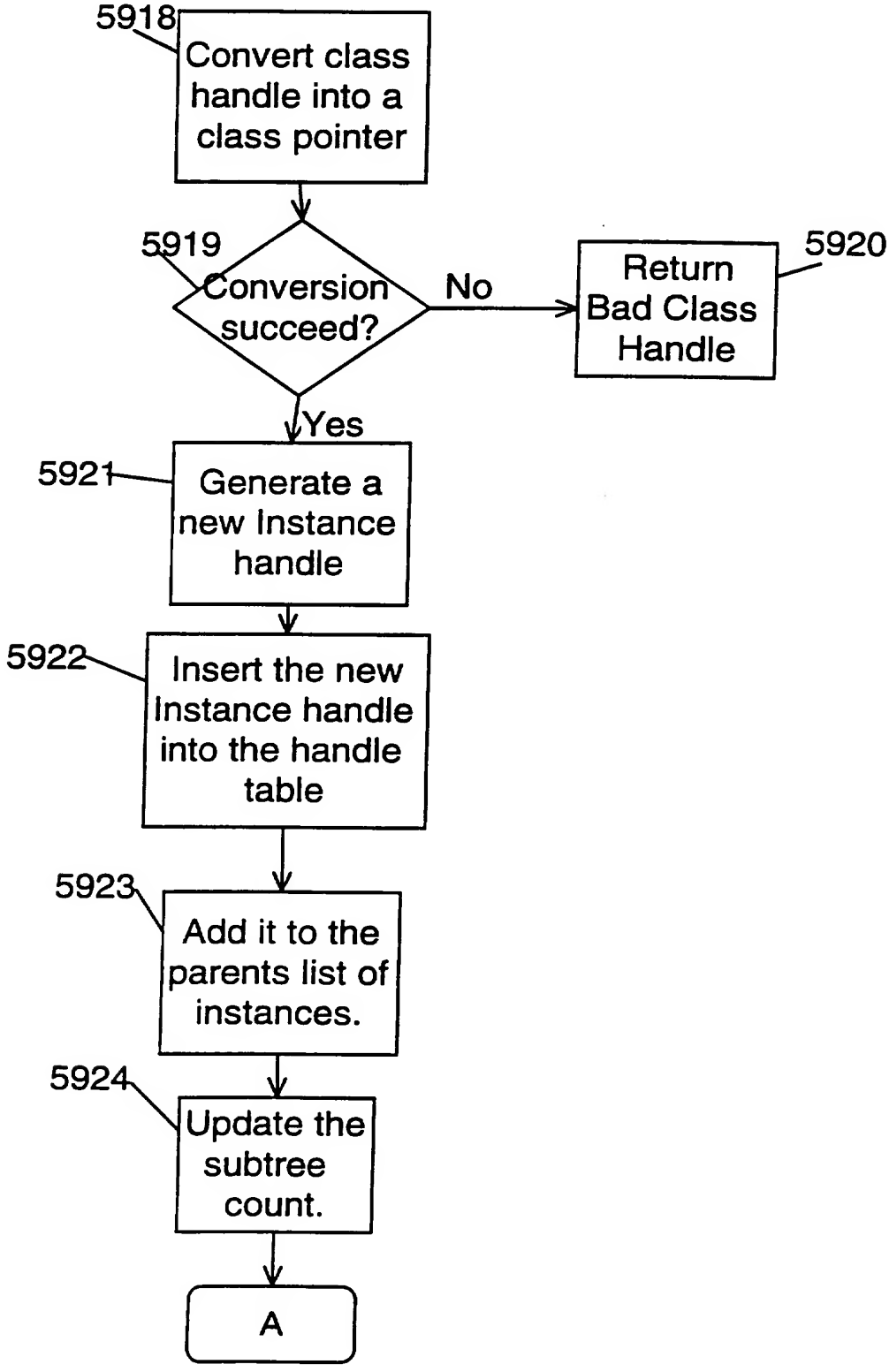

5918
Convert class handle into a class pointer
5919
Conversion succeed?
No
5920
Return Bad Class Handle
Yes
5921
Generate a new Instance handle
5922
Insert the new Instance handle into the handle table
5923
Add it to the parents list of instances.
5924
Update the subtree count.
A

FIG. 284

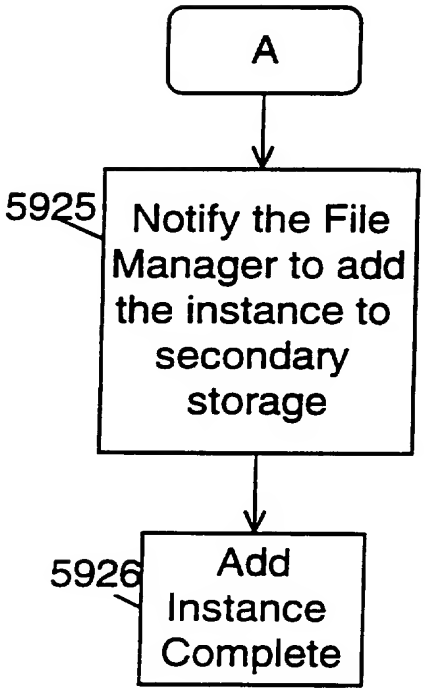

A
5925
Notify the File Manager to add the instance to secondary storage
5926
Add Instance Complete

FIG. 285

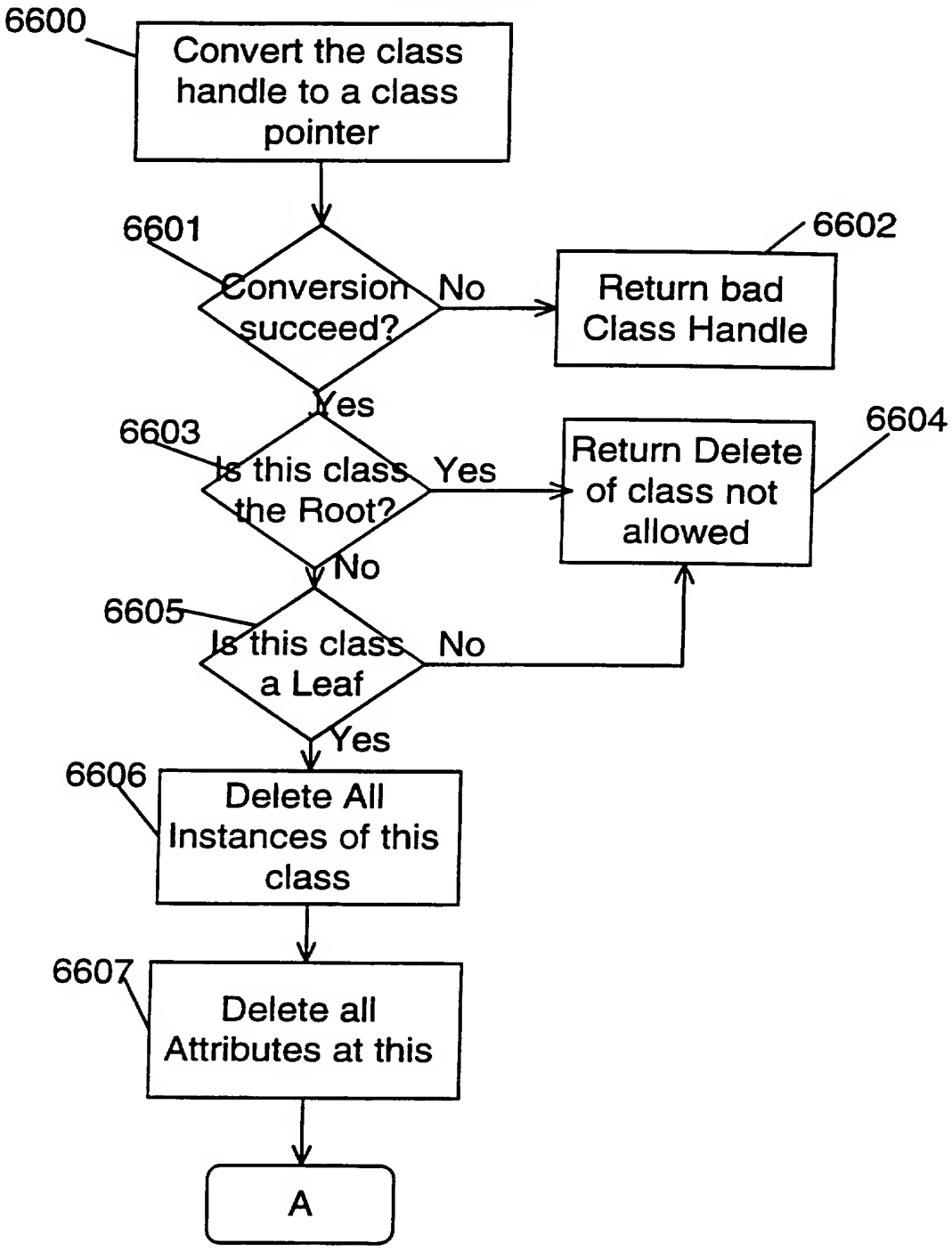

6600
Convert the class handle to a class pointer
6601
Conversion succeed?
No
6602
Return bad Class Handle
Yes
6603
Is this class the Root?
Yes
6604
Return Delete of class not allowed
No
6605
Is this class a Leaf
No
Yes
6606
Delete All Instances of this class
6607
Delete all Attributes at this
A

FIG. 286

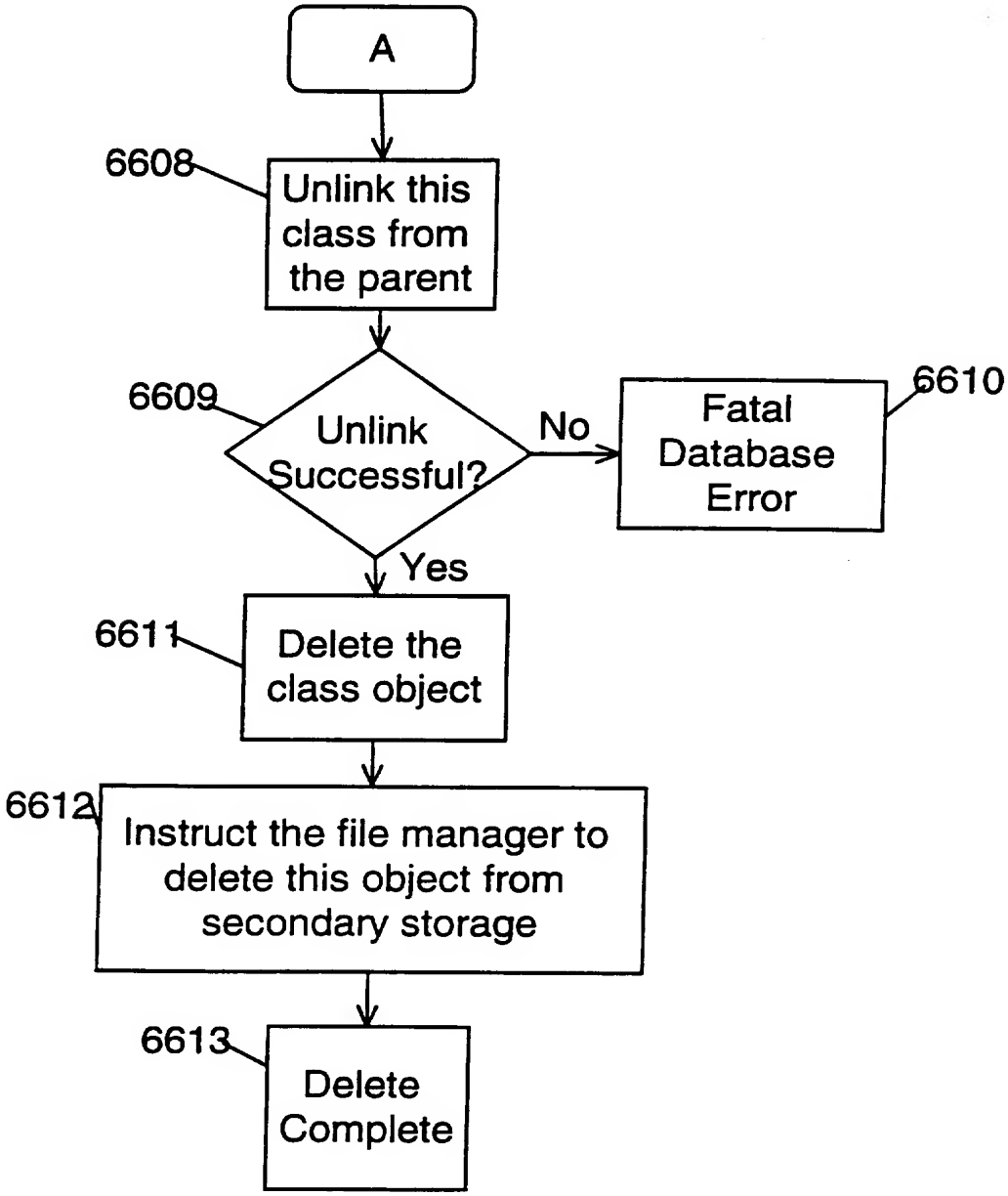

A
6608
Unlink this class from the parent
6609
Unlink Successful?
No
6610
Fatal Database Error
Yes
6611
Delete the class object
6612
Instruct the file manager to delete this object from secondary storage
6613
Delete Complete

FIG. 287

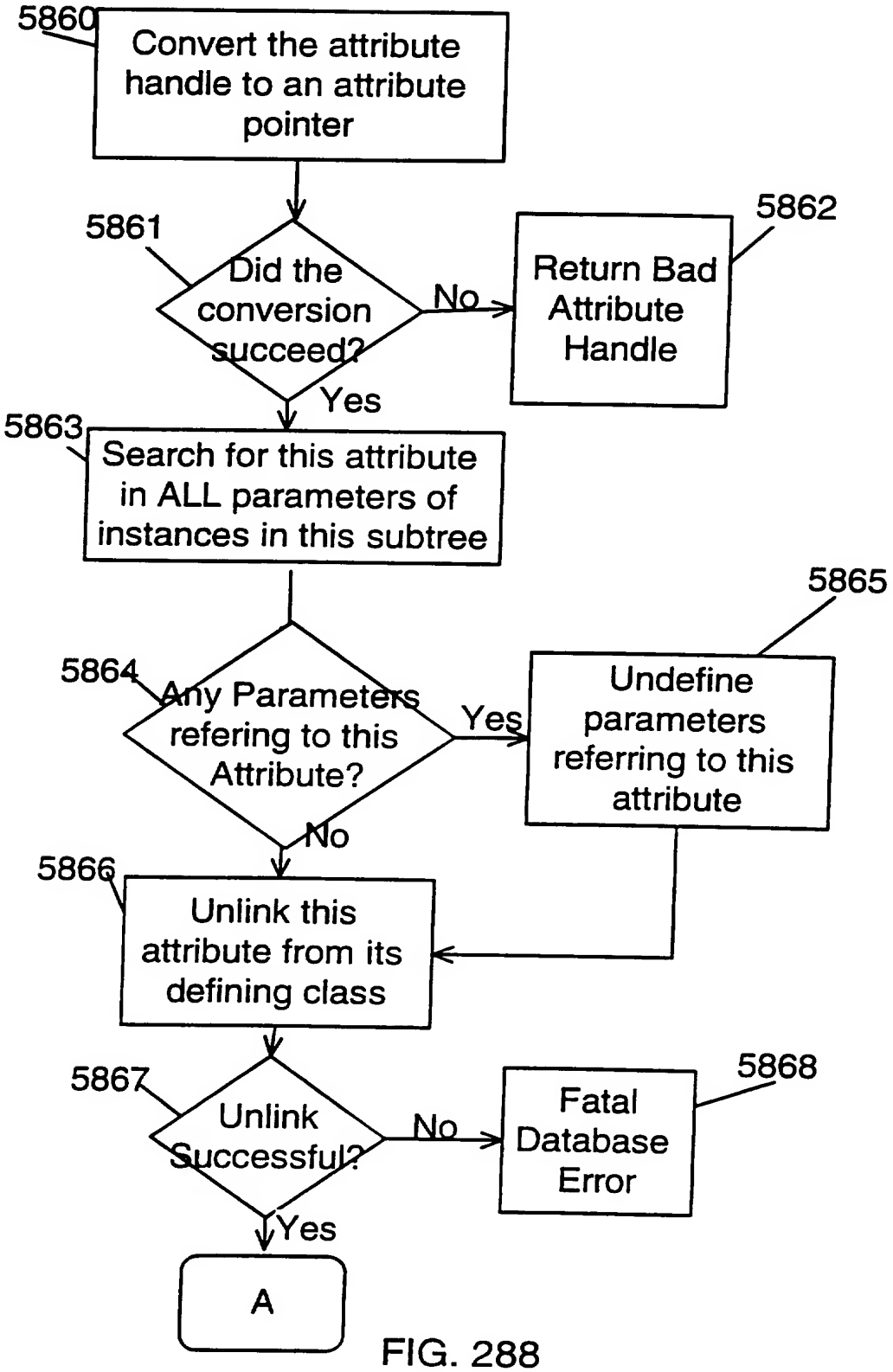

5860
Convert the attribute handle to an attribute pointer
5861
Did the conversion succeed?
No
5862
Return Bad Attribute Handle
Yes
5863
Search for this attribute in ALL parameters of instances in this subtree
5865
5864
Any Parameters refering to this Attribute?
Yes
Undefine parameters referring to this attribute
No
5866
Unlink this attribute from its defining class
5867
Unlink Successful?
No
5868
Fatal Database Error
Yes
A

FIG. 288

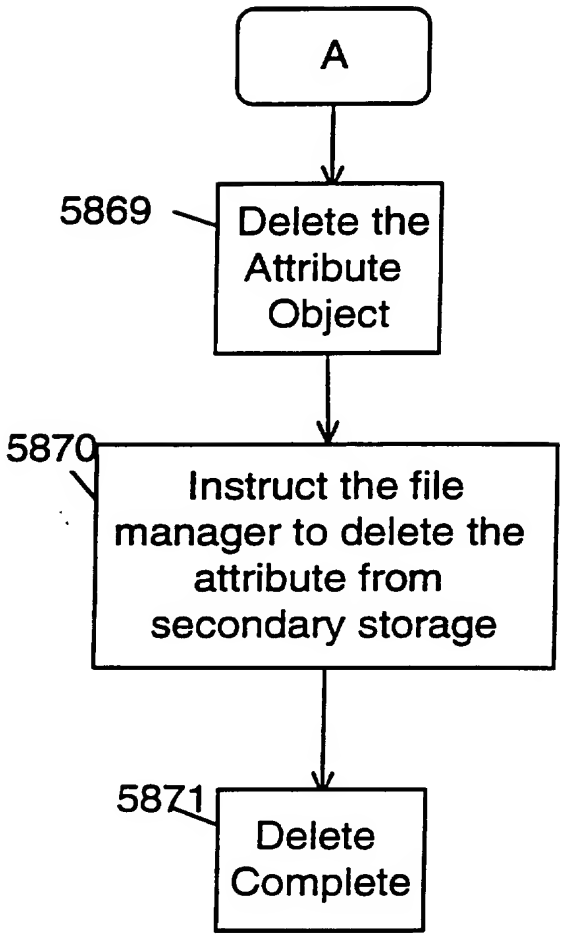

A
5869
Delete the Attribute Object
5870
Instruct the file manager to delete the attribute from secondary storage
5871
Delete Complete

FIG. 289

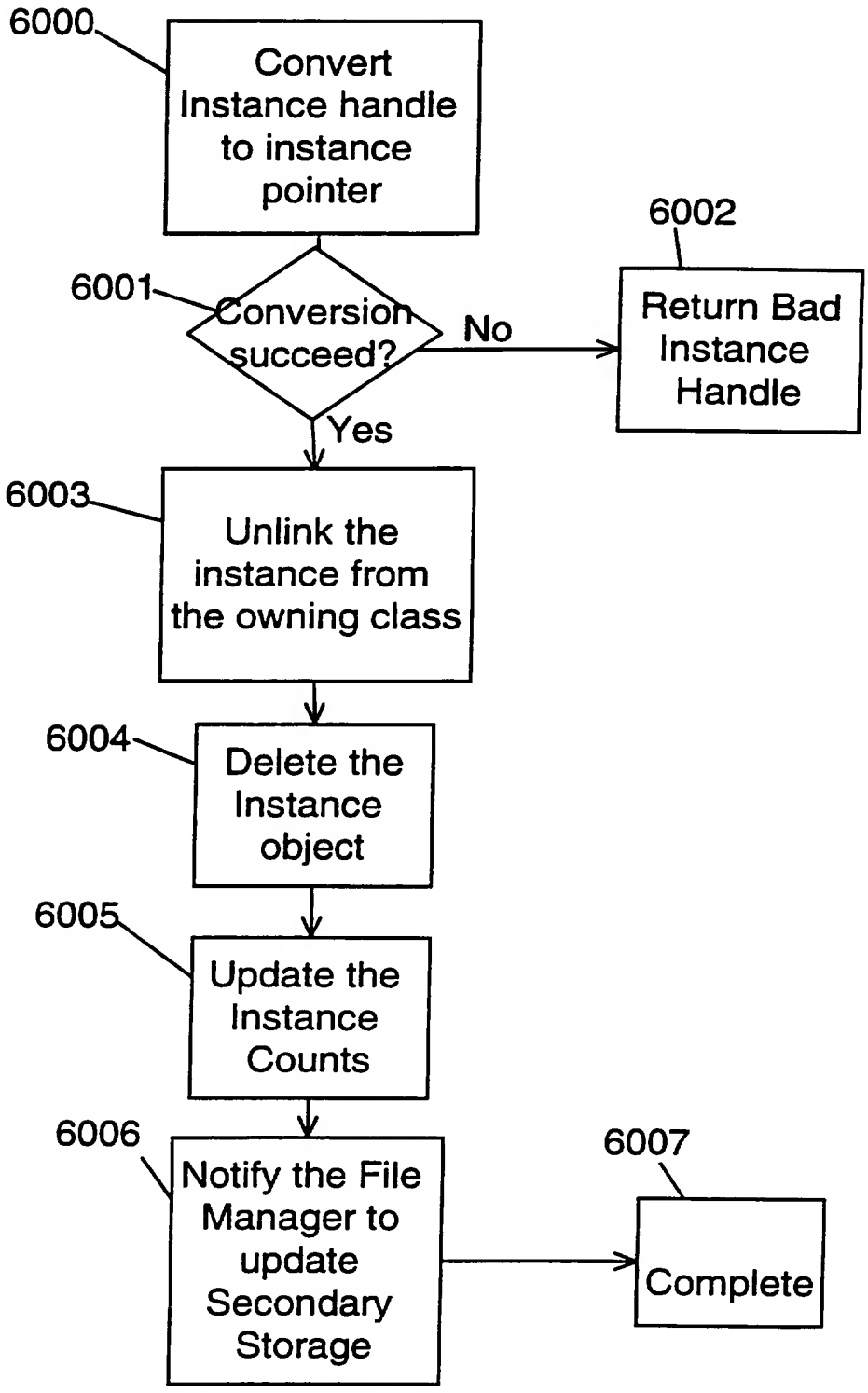

6000
Convert Instance handle to instance pointer
6001
Conversion succeed?
No
6002
Return Bad Instance Handle
Yes
6003
Unlink the instance from the owning class
6004
Delete the Instance object
6005
Update the Instance Counts
6006
Notify the File Manager to update Secondary Storage
6007
Complete

FIG. 290

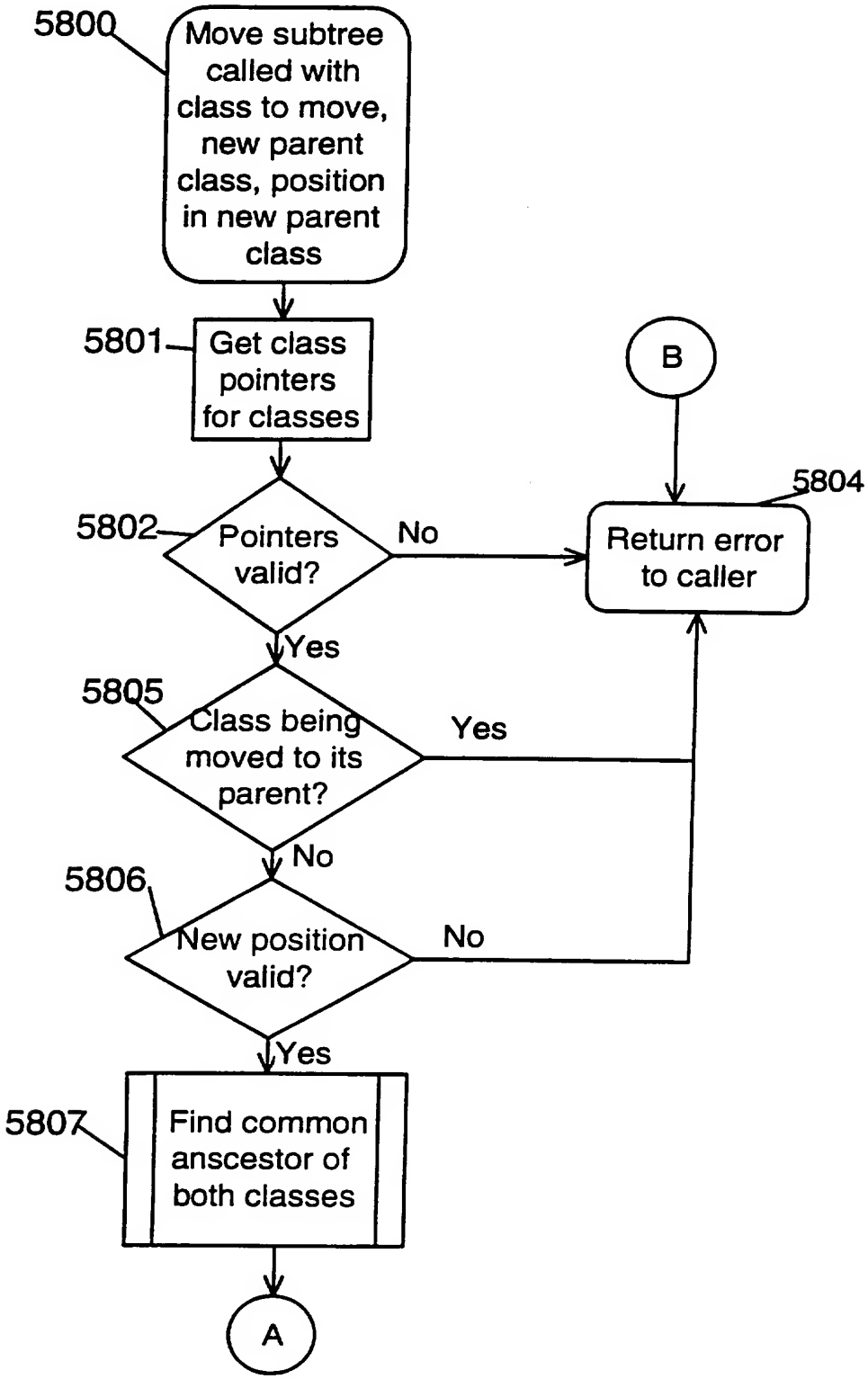

5800
Move subtree called with class to move, new parent class, position in new parent class
5801
Get class pointers for classes
B
5804
5802
Pointers valid?
No
Return error to caller
Yes
5805
Class being moved to its parent?
Yes
No
5806
New position valid?
No
Yes
5807
Find common anscestor of both classes
A

Fig. 291

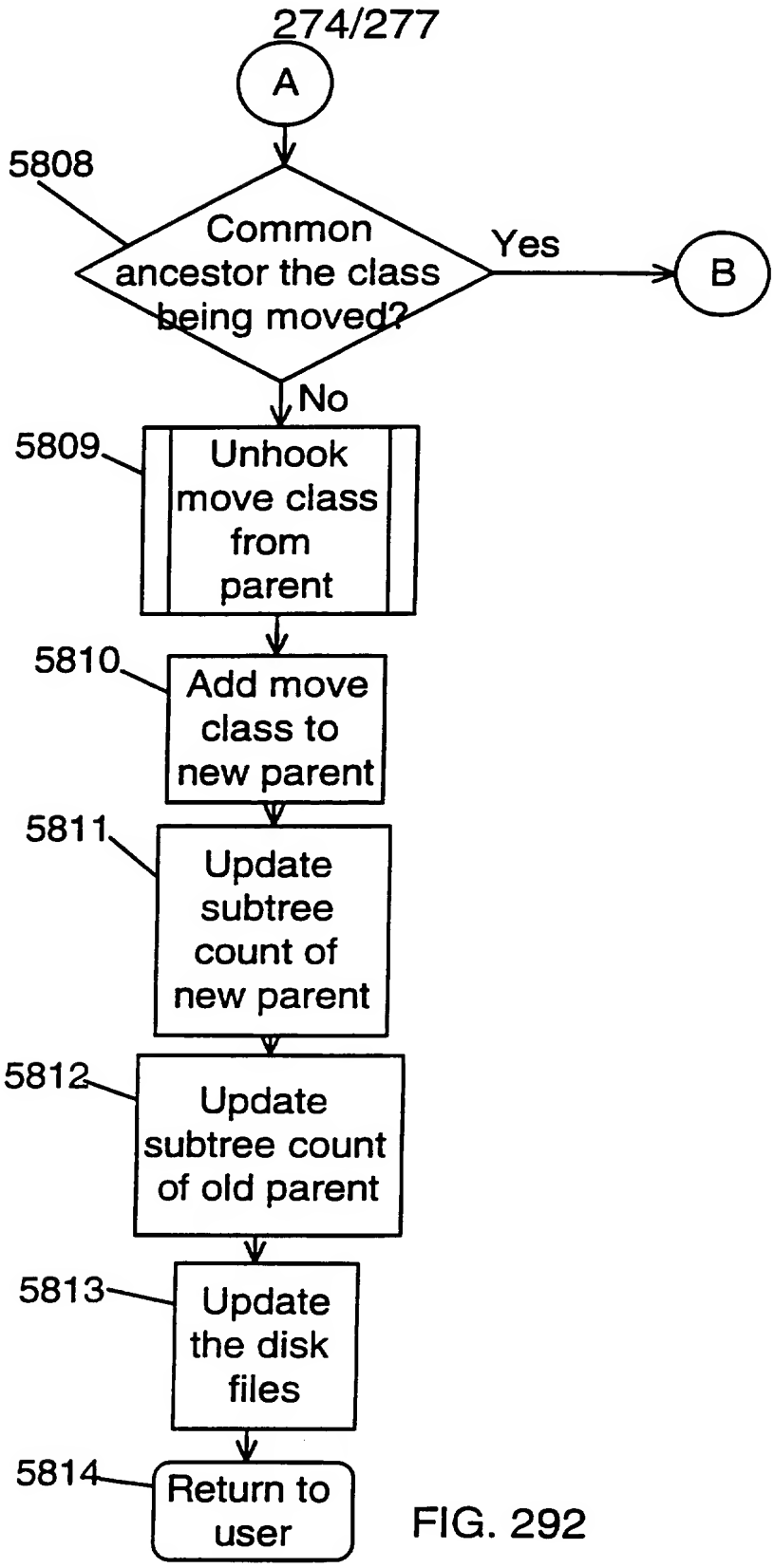

274/277
A
5808
Common ancestor the class being moved?
Yes
B
No
5809
Unhook move class from parent
5810
Add move class to new parent
5811
Update subtree count of new parent
5812
Update subtree count of old parent
5813
Update the disk files
5814
Return to user

FIG. 292

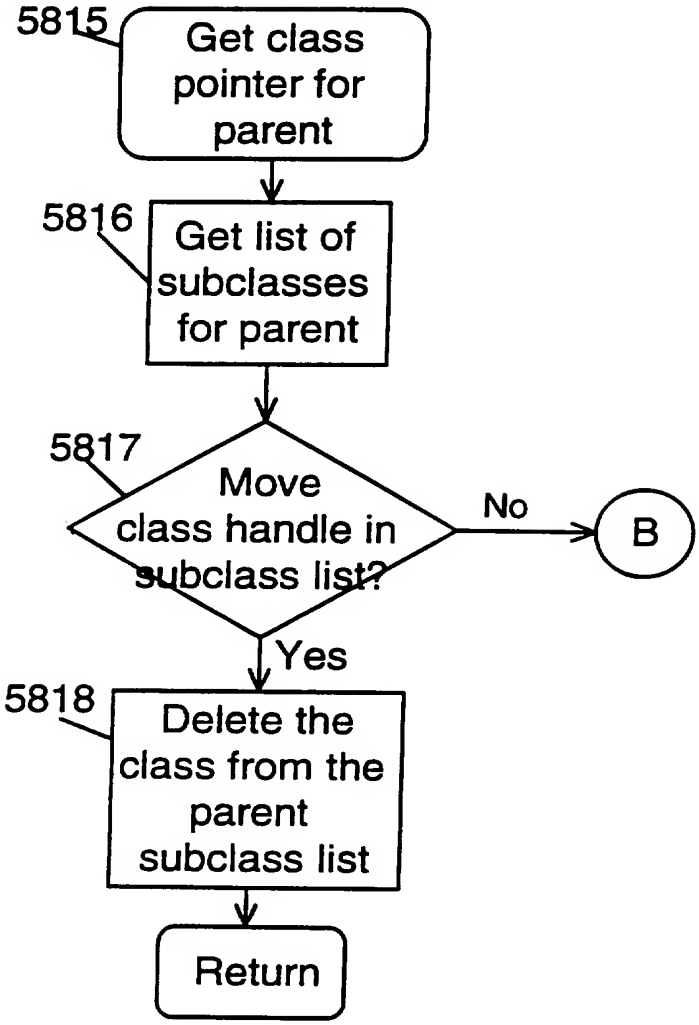

5815
Get class pointer for parent
5816
Get list of subclasses for parent
5817
Move class handle in subclass list?
No
B
Yes
5818
Delete the class from the parent subclass list
Return

FIG. 293

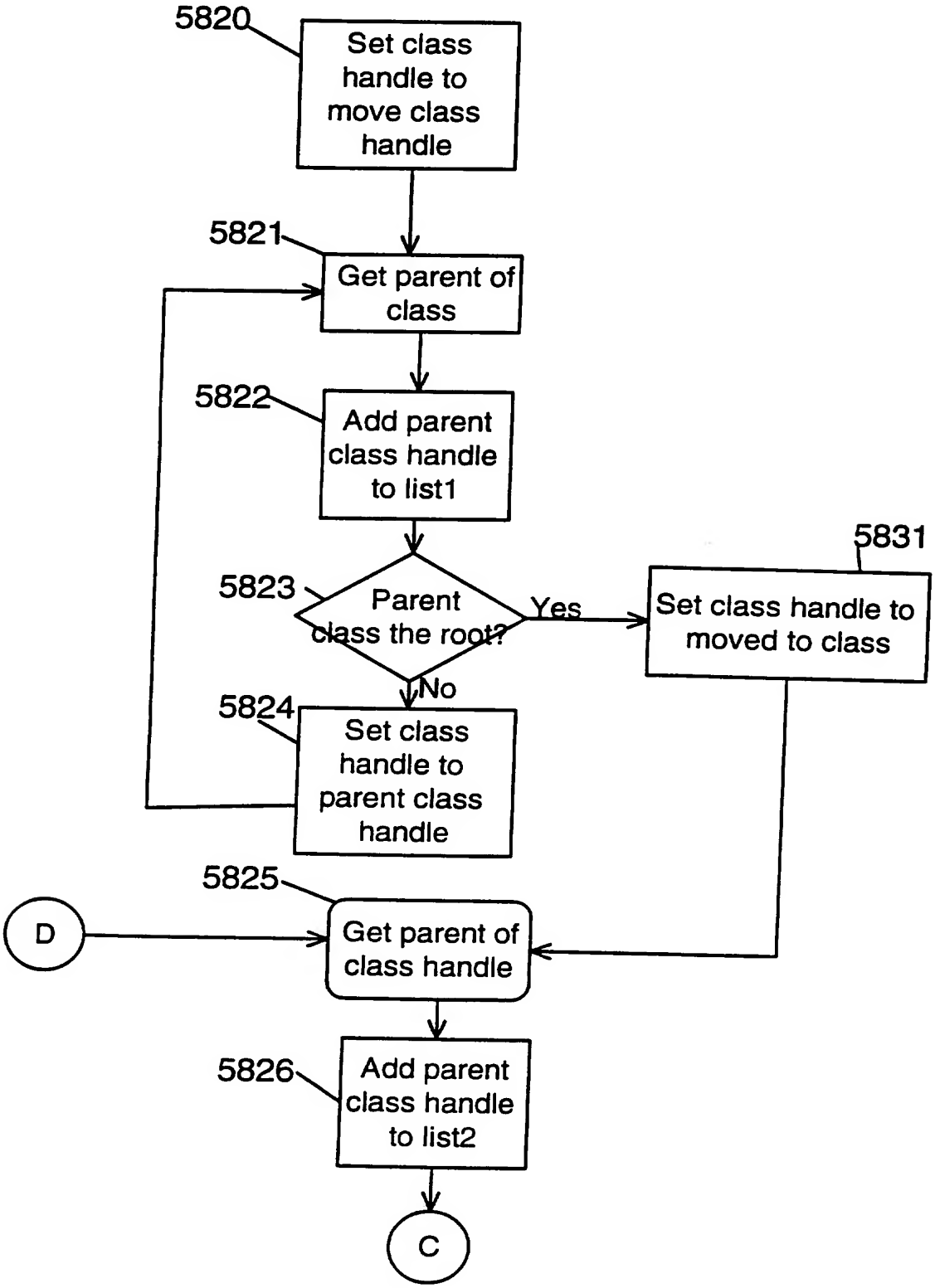

5820
Set class handle to move class handle
5821
Get parent of class
5822
Add parent class handle to list1
5823
Parent class the root?
Yes
5831
Set class handle to moved to class
No
5824
Set class handle to parent class handle
5825
D
Get parent of class handle
5826
Add parent class handle to list2
C

FIG. 294

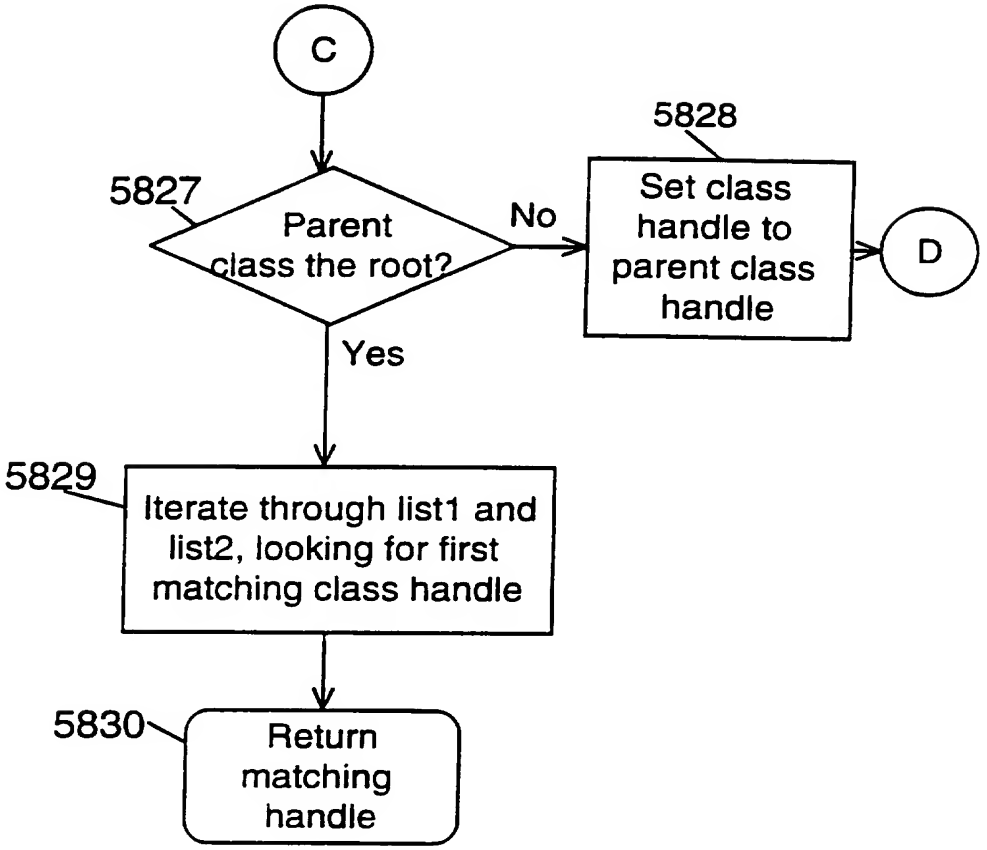

C
5828
5827
Parent class the root?
No
Set class handle to parent class handle
D
Yes
5829
Iterate through list1 and list2, looking for first matching class handle
5830
Return matching handle

FIG. 295

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US95/15028

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) :G06F 17/30
US CL : 395/600

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 395/600, 700

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

APS search terms object oriented, database, knowledge base, handle, instance, class, attributes, legacy

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US, A, 4,930,071 (TOU ET AL) 29 MAY 1990	
A	US, A, 5,133,075 (RISCH) 21 JULY 1992	
A	US, A, 5,021,992 (KONDO) 04 JUNE 1991	

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

*	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be part of particular relevance	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"E"	earlier document published on or after the international filing date	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"&"	document member of the same patent family
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search
17 APRIL 1996

Date of mailing of the international search report
22 APR 1996

Name and mailing address of the ISA/US
Commissioner of Patents and Trademarks
Box PCT
Washington, D.C. 20231
Facsimile No. (703) 305-3230

Authorized officer
JOHN C. LOOMIS for
Telephone No. (703) 305-3833

Form PCT/ISA/210 (second sheet)(July 1992)*